Pulse Coupled Neural Networks for the Segmentation of Magnetic Resonance Brain Images

THESIS

Shane Lee Abrahamson
First Lieutenant, USAF

AFIT/GCS/ENG/96D-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/96D-01

Pulse Coupled Neural Networks for the Segmentation of Magnetic Resonance Brain Images

THESIS

Shane Lee Abrahamson
First Lieutenant, USAF

AFIT/GCS/ENG/96D-01

Approved for public release; distribution unlimited

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

AFIT/GCS/ENG/96D-01

Pulse Coupled Neural Networks for the Segmentation of Magnetic Resonance Brain Images

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Science

Shane Lee Abrahamson, B.S.

First Lieutenant, USAF

December, 1996

Acknowledgements

There are several people I would like to thank that contributed so much to this research. First Dr. Steven Rogers, whose guidance and mentoring led me to explore the wonderful world of computer vision and image analysis. I also thank Dr. Matthew Kabrisky and Dr. Martin DeSimio for their invaluable instruction and insight. Finally, thanks to Captain Randy Broussard for spending endless hours familiarizing me with pulse coupled neural networks as well as providing many of the graphics which are contained in this research.

I would also like to thank my parents for their endeavor to instill in me a sense of purpose and for igniting the desire to seek out the knowledge that interests me. I thank you for allowing me from an early age to make my own decisions.

I would like to dedicate this research to my wife Andrea and to our children Annika and Alex. This research would have been impossible without Andrea's support and encouragement. My work and responsibilities were your sacrifices and I am forever grateful. To precious little Annika, your proclamations of my arrival home each night lifted my spirits and served as a constant source of motivation. And Alex, I thank you for forcing me away from my work and giving me those much needed 'breaks.' I couldn't be more proud.

Shane Lee Abrahamson

Table of Contents

List of Figures

Abstract

This research develops an automated method for segmenting Magnetic Resonance (MR) brain images based on Pulse Coupled Neural Networks (PCNN). MR brain image segmentation has proven difficult, primarily due to scanning artifacts such as interscan and intrascan intensity inhomogeneities. The method developed and presented here uses a PCNN to both filter and segment MR brain images. The technique begins by preprocessing images with a PCNN filter to reduce scanning artifacts. Images are then contrast enhanced via histogram equalization. Finally, a PCNN is used to segment the images to arrive at the final result. Modifications to the original PCNN model are made that drastically improve performance while greatly reducing memory requirements. These modifications make it possible to extend the method to filter and segment three dimensionally. Volumes represented as series of images are segmented using this new method. This new three dimensional segmentation technique can be used to obtain a better segmentation of a single image or of an entire volume. Results indicate that the PCNN shows promise as an image analysis tool.

Pulse Coupled Neural Networks for the Segmentation of Magnetic Resonance Brain Images

I. Introduction

1.1 Introduction

Current technology enables the detection, diagnosis, and evaluation of many common and not so common ailments through non-invasive imaging. One of these imaging techniques is magnetic resonance imaging (MRI). Through this procedure, physicians obtain images, or 'slices', of various parts of the human body. They then analyze these slices in an attempt to gather information to perform a variety of tasks. One anatomical region of particular interest is the human brain. MRI permits researchers to study the delicate brain's functional and physical characteristics with little risk to the patient.

Presently, MRI is able to generate hundreds of images of an anatomical region (head, thorax, abdomen, etc.). For diagnostic purposes, it is the physician's responsibility to examine the slices and identify regions of interest (ROI). The ROI is then divided into its constituent anatomical structures and possible anomalies. This human division of regions is analogous to computer segmentation of images. With the profusion of medical imaging data available to the physician, automated methods have been suggested as a means of easing the burden and allowing him/her to focus on the tasks mentioned.

MR image segmentation is an important step for a number of applications that include the identification of anatomical ROIs for diagnosis, treatment, or surgery planning, preprocessing for multimodality image registration, and tumor volume measurement. Application has also been proposed for the diagnosis of brain trauma and multiple sclerosis. These applications present significant problems [3].

Medical image segmentation is not a solved problem. Various methods have been applied, including neural network and statistical theory based approaches [3]. However, image and tissue properties have prevented methods from reliably producing good results.

Recently, a new neuronal model was developed based on the primate visual system [9]. A network founded on this model is the pulse coupled neural network. This model shows promise because it may model the neuron's behavior more closely than other popular neural networks. However, this new model has not been widely applied to real world problems to determine its effectiveness.

1.2 Problem Statement

This research will adapt, implement, and test a pulse coupled neural network for segmenting two-dimensional MR images.

1.3 Scope and Assumptions

The medical data used in this research is magnetic resonance head data. The data was obtained from Armstrong Laboratories and consists of images of normal human volunteers. Each image is a 256 x 256 array of gray scale pixel intensity values.

The scope of this thesis is to investigate the application of a pulse coupled neural network (PCNN) model to the segmentation of MR brain images and volumes.

1.4 Thesis Organization

The following chapter provides background information describing the MR imaging process. The chapter also includes a discussion on image segmentation methods along with different image preprocessing techniques. Chapter III describes the image acquisition method, the MR image filtering method, the segmentation method, and the adaptation of the pulse coupled neural network. Chapter III also describes the results of segmenting the MR brain images and volumes. The conclusions and recommendations are given in Chapter IV.

II. Background

2.1 Introduction

This chapter provides an introduction to Magnetic Resonance Imaging (MRI), an evaluation of segmentation techniques, and background on the pulse coupled neural network. Particularly, it emphasizes background material necessary to understand an MR image and the current state of MR image segmentation techniques.

2.2 Magnetic Resonance Imaging (MRI)

MRI uses magnetism and radio frequencies (RF) to create diagnostic sectional images of the body [2]. The first magnetic resonance image was produced in 1972 with the first diagnostic human images being produced by 1977.

The spinning nucleus of an atom is a charged piece of matter that creates a magnetic field. When the nuclei generating these fields are placed within another magnetic field, the nuclei align themselves with the field. Since the spinning nuclei have mass, they also have moments of inertia. The moments resist changes in angular momentum and thus attempt to inhibit the nuclei from aligning in a magnetic field. The ratio of magnetic moment to the moment of inertia is called the gyromagnetic ratio. Each isotope has a specific ratio, allowing it to be identified during MR imaging [2].

Since each gyromagnetic ratio is specific to its isotope, it was hoped that the ratio along with the length of relaxation time after an RF pulse would allow the distinction between healthy and diseased tissue. However, the determination of relaxation times for molecules has proven too complex and MRI data alone can not diagnose tissue type. The value of MRI relies heavily on the radiologist's visual diagnostic abilities [2].

2.3 Current Technology

This section presents preprocessing and segmentation methods that have been applied to MR brain images. Each of the methods is reviewed and its merits and problems noted. An understanding of the approaches taken thus far gives insight into developing

new approaches with the hope of avoiding past pitfalls and capitalizing on strengths as well as providing a baseline for comparing the results of this research.

Figure 2.1 illustrates the basic components of a generic image processing system [3]. As can be seen, image segmentation is but one of the pieces in such a system, and is not typically the first. It is usually preceded by image preprocessing and feature extraction and *can be* followed by a classification step which assigns labels to the segmented regions. Such an illustration provides a good context for the review of the state of image segmentation. Each step represents the performance of a unique function in the image analysis system. Not all systems perform every function. The steps a system performs and the manner in which it performs them define or characterize the system. Therefore, each of these steps will be discussed in succession with existing methods' approaches to the performance of such steps presented.

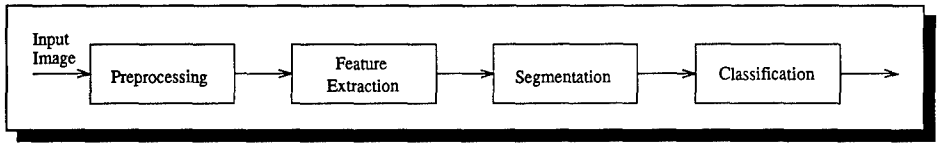


Figure 2.1 Components of an image analysis system.

2.3.1 Image Preprocessing. Image preprocessing can be the first step in image understanding, and before applying any of the segmentation methods, preprocessing can be performed in an attempt to improve the segmentation results. Two preprocessing techniques applicable to image segmentation in general are noise removal and contrast enhancement. Noise removal and contrast enhancement techniques commonly applied to single gray scale MR images include linear low pass filtering, adaptive filtering, and nonlinear adaptive filtering. Specific to MRI, radio frequency nonuniformity corrections have also been attempted [3]. Preprocessing techniques try to reduce the artifacts introduced by the imaging modality. Example artifacts are random noise, partial volume effects, and intensity inhomogeneities due to the nonuniform radio frequency fields of the MRI scanner.

Traditional linear low pass filtering reduces image noise, however, edge definition is not maintained. Image blurring and loss of fine detail occurs. Therefore, the cost for noise

reduction is loss of spatial resolution which can lead to erroneous segmentation results. The process developed by Tsai [1] uses low pass filtering.

Adaptive filtering tries to overcome the widely varying local intensity distributions that occur in different spatial locations in the image. Adaptive filtering techniques include Bayesian image restoration, wavelet analysis, and anisotropic diffusion filtering. Bayesian image restoration has shown promising results for noise reduction and edge enhancement, but is sensitive to parameter settings and requires long computation times. Wavelet analysis has also demonstrated good results for noise removal and edge detection. However, while wavelet techniques have the advantage of local scale-space encoding, some fine detail is lost during processing and it is unclear how to set the wavelet coefficient reduction values.

Some nonlinear adaptive filtering techniques have been reported in the literature. These techniques report some good results, however, they aren't without their difficulties. One method reports impressive signal-to-noise ratios which aren't computed according to MR conventions, therefore, leaving the results suspect. Another approach introduces spike effects into the images.

Therefore, an ideal image preprocessing method applied to medical images should reduce image noise while maintaining the fine detail and preserving edges. A robust method that is insensitive to operator settings and can overcome imaging modality artifacts is desirable. So, while preprocessing is an important step in the overall analysis process, not all reported methods perform preprocessing outright at the beginning of the process. This does not mean that it is not performed. These methods try to account for image artifacts during the segmentation process itself.

2.3.2 Feature Extraction. MRI segmentation is based on a set of measurable features which are extracted or computed from the images. Features themselves can be classified as pixel intensity-based features, calculated pixel intensity-based features, and edge and texture-based features.

Segmentation approaches that rely solely on the gray scale intensity values of the pixels themselves as features are simply intensity-based. Intensity values can be from a single image, a volume data set, a multispectral data set, or a multimodal image set.

When performing pixel-based segmentation, each pixel in the image has a corresponding feature, typically intensity-based. However, when dealing with multispectral data, each slice will have multiple images, each of different contrast. Some methods use a set of feature vectors, each containing a number of features equal to the number of different images used [7] [5].

One problem when dealing with multispectral MRI is the selection of features so that tissue differentiation is maximized while computational complexity is minimized. Multimodal data sets may also have to be registered prior to segmentation to arrive at meaningful results. Otherwise, a given feature vector may contain feature values not corresponding to that particular sample.

Features derived using calculated MR imaging parameters are calculated features. These features appear enticing since knowledge of the image acquisition method and MR system parameters could lead to better tissue classification. However, the nonlinear nature of the calculations involved introduces noise that makes the differentiation of tissues and segmentation reproducibility problematic. An adaptive MRI segmentation technique using calculated features showed promising results [13]. It was based on an estimated gain field that multiplied the image intensity data in an attempt to correct the intrascan intensity inhomogeneities due to the inherent problems with MR image acquisition. Another approach, the gray white decision network (GWDN), used two types of input with the first calculated from the pixel intensities. Image pixel intensities are passed through similarity functions for each tissue class. The results of these calculated measures are one of the features input to the network.

Edge and texture-based features are yet more types of image features. Although there are several methods for extracting edges from images, none have yet proven very reliable for MRI segmentation. Typically, they are used in conjunction with other types of features. Noted earlier, the GWDN relies upon two types of features. The second feature

type is edge information. Texture, however, is a statistical feature derived from a number of pixels. Texture features have typically been applied to tissue classification rather than image segmentation.

2.3.3 Segmentation. Image segmentation groups pixels into regions, and therefore defines object regions [3]. Segmentation uses the features extracted from the image(s), therefore, good feature selection greatly influences segmentation results.

Segmentation methods are not as easily categorized as features. The various aspects of the segmentation process really define it. There are no broad generalizations that can be made about the methods. In other words, it cannot be said that certain methods are single image methods and others are multispectral methods or that some methods are supervised and others are unsupervised.

A problem with all segmentation methods is the amount of human operator supervision that is required. The effect that operator involvement has on segmentation accuracy and reproducibility is of paramount concern, especially when dealing with critical systems. The imaging method itself can inject nonuniformity and noise that must be considered to obtain accurate segmentation. Segmentation validation is another problem. The lack of absolute ground truth for some imaging modalities precludes the objective evaluation of segmentation results.

MRI segmentation methods use either a single 2D or 3D image or a series of multispectral or multimodal images. Each multispectral image represents the same physical location but with a different contrast. Common segmentation approaches to MR images are thresholding, edge detecting, clustering, genetic algorithms, neural networks, and probabalistic techniques.

The most obvious form of a threshold-based segmentation method is the use of a global threshold. The difficulty lies in determining the value of the threshold. Again, because of intensity inhomogeneities in the images and overlapping class intensities, global thresholding methods themselves have not been very successful. To achieve reasonable segmentation results, other techniques or variations have been applied in conjunction with

global thresholding. These include knowledge-guided methods using 'goodness functions', local instead of global thresholding, and morphological filtering.

A process developed by Tsai [1] does exactly this. Thresholding is used to extract all brain tissue from surrounding background noise. The accumulated image histograms are used to compute the threshold. The first local minimum of the histogram becomes the threshold. Morphological operations then extract only the brain matter. Thresholding is further used to segment the brain into regions.

Edge detection methods are another approach to image segmentation. These methods are susceptible to image noise, over and under segmentation, and the variability of the threshold required to select edges in an image. However, one method uses the oversegmentation nuisance to its advantage. A genetic algorithm based method [8] actually performs edge-based region growing to obtain oversegmentation.

Another edge-based method was developed to locate cortical convolutions in MR brain images. The method of [4] seeks only to isolate the brain matter from the rest of the image. The process begins with an initial polygon estimation of the contour of the brain object. This contour is drawn manually by a human technician. Once the contour is obtained, a perpendicular transform of the image takes place. Perpendiculars along each edge of the polygon contour are computed and the image is resampled along these using bilinear interpolation. A transformation matrix contains a row of intensity values corresponding to the intensities of each resampled perpendicular. One-dimensional morphological opening is then applied to each row of the transform matrix. Each pixel in the transformation matrix is then assigned a cost according to a cost function that is defined over a number of terms that measure predefined relevant properties. Dynamic programming is then used to compute a minimum cost path in the matrix. Pixels in the transformation matrix belonging to the path are mapped to the original image to form a new contour. The process repeats with the new contour as the starting point. The processing ends when the difference between successive contours meets a threshold.

When processing proceeds to a new slice, the contour obtained from the previous slice is the initial contour for the new slice. However, if the contour is not satisfactory, the

technician may enter a new contour. This indicates the need for constant supervision and possible human intervention.

The previous process shows that edge detection methods typically require operator region editing to achieve good results. A method similar to edge detection is boundary tracing. Boundary tracing consists of an operator choosing a pixel in a region and the method then finds the boundary of that region and follows the boundary. This method is more amenable to segmenting well defined structures since a good guess of the initial boundary is required.

Clustering segmentation methods attempt to find the natural gathering areas of image or pixel features. Pixels or objects that are near a given cluster are assigned to that cluster. A method presented by [5] performs MR image segmentation based on clustering and compares the results for normal and diseased tissues. The approaches use fuzzy c-means algorithms as their basis for segmentation.

A fuzzy *c*-means (FCM) method follows a fuzzy c-means model which prototypes the functions used to derive values used in the algorithm [5]. The algorithm is unsupervised and works in the following manner. For every feature vector, the algorithm computes a class membership value for every class for that vector using some initial class cluster estimates. The process then iteratively computes class clusters based on membership values and then computes a new set of membership values. If the maximum difference between old membership values and new values is less than some threshold, the algorithm terminates with the new values.

An adaptation to the fuzzy *c*-means algorithm (AFCM) is to use approximations for some of the calculations [5]. This is done primarily for accelerating the completion of the algorithm. This modification provides approximately one order of magnitude performance increase. However, the modification may cause the adapted fuzzy *c*-means algorithm to terminate with different results than the original method.

Yet another approach to MR image segmentation is neural networks. Neural networks attempt to simulate the physiological behavior of a neural connection matrix. Features are typically fed as inputs to the networks and the outputs are the classification of the

representative samples. Neural networks typically require training and their architectural constructions vary.

One neural-network based segmentation technique developed attempts to overcome the inter-slice intensity variations of MR images through an adaptive learning scheme [7]. The artificial neural network (ANN) requires training on samples selected from each of the classes and its segmentation performance compared to a maximum likelihood classifier (MLC) is dependent upon the ANN's training set size. However, the performance of the ANN is relatively insensitive to the selection of training sets. In other words, even if the training sets do not appropriately sample the image, the ANN performed reasonably better than the MLC in the same situation.

The ANN used in [7] was then modified to segment in three dimensions. The research clearly illustrates that the tissue classes in MR brain studies are not linearly separable and that the problem is magnified from slice to slice. However, the ANN adaptive algorithm exploited the fact that there is some continuity of structures from slice to slice and that a classifier trained on a given slice can be expected to perform reasonably well on the slice directly above and below. The process works by training an ANN classifier on user selected training samples from a given slice and then segmenting that slice. Next, the slice directly above or below the segmented slice is classified using the trained ANN. The slices are then superimposed and corresponding pixels that received the same classification form a training set for the slice directly above or below. Finally, the ANN is retrained using the new training set.

A Hopfield neural network was also used to segment MR images [12]. Typical ANNs such as generalized perceptrons and back propagation networks are unlike supervised parametric methods in that they make no use of *a priori* probability distributions. However, they do require training sets and therefore require supervision similar to parametric methods. The Hopfield net used, however, is unsupervised and seeks to classify the images in such a way as to minimize an energy function. The energy function expresses the current classification of image pixels as a sum of distances. The distances being the weighted distances between a pixel intensity and the centroid of the class to which the pixel is currently

assigned. Inputs to neurons are changed by the weighted difference. The energy function is minimal when the input-output activity of the neurons approaches zero.

However, because the method only accounts for pixel intensity and does not include any spatial information during classification, the resulting segmented images must be postprocessed. A majority filter was used to eliminate the speckle artifacts present after classification. Depending upon the size of the filter window, the resulting image may lose small details.

An analogue constraint satisfaction neural network was constructed to segment MR brain images [15]. The network developed is termed the Grey-White Decision Network (GWDN), so called for its desire to distinguish grey and white brain matter. The GWDN differs from the Hopfield net development in the way that the network's architecture was developed. Instead of developing a network's architecture from an energy function, it is developed from the defined task constraints. Task constraints may be conflicting for an optimal solution. For example, one constrain might be that pixels of a tissue type tend to be near other pixels of that type. Another constraint might be that pixel intensities cluster about certain average values. These constraints will conflict if a pixel intensity is that of a tissue average but located amidst pixels of a different class. The GWDN attempts to handle this constraint conflict problem.

The GWDN contains a number of neural layers equivalent to the number of desired tissue classes. Each layer corresponds to a particular tissue class and each neuron corresponds to a pixel in the original image. The layers compete for pixel classification and there is only one active neuron at any pixel location. This is accomplished through inhibiting connections between layers while cooperation between neurons within layers strengthens groups of similar pixels. The cooperation within layers is performed by excitatory connections and reciprocal connections from each neuron to itself allow it to sustain once activated.

When a strong edge is detected between two pixels in an image, the excitatory connections between the two corresponding neurons in each layer are reduced or shut off. However, the inhibitory connections between layers are not affected.

The resulting network works in the following manner. After initialization of the neurons to zero, image and edge input is provided to the layers of the network. Neurons with intensities close to the peak of their similarity functions will begin to activate. Once neurons are activating, they will attempt to inhibit the activation of neurons of different layers while at the same time try to excite its neighbors in the same layer into activation. Once equilibrium is achieved, neurons with the highest activation 'win' and the pixel is classified according to the neuron's tissue layer.

It is noted in [15] that certain conditions can arise causing a network of this type to become unstable and produce oscillations precluding the network from attaining the equilibrium noted earlier. This problem can be overcome by modifying the network in the following way. Instead of allowing active neighbors to contribute to the excitatory terms of neurons, have them decrease the inhibition contributions from surrounding layers.

One of the major drawbacks of the GWDN is the requirement for the tissue class means and standard deviations used to compute Gaussian similarity functions. However, this cannot be realized unless there is some truth data available or the image data is already segmented. In reality, even a segmented image is only an approximation and cannot be proven to correctly classify all intensities. An alternative is to estimate the necessary parameters from entire image histograms or from image regions.

Another method employed in [5] is a feedforward cascade correlation neural network (FFCC). The FFCC used is a supervised method attempting to overcome the speed limitation of a feedforward back-propagation network and to allow incremental learning [5].

The cascade architecture works in the following manner. Initially, the network consists of an input layer and an output layer determined by the problem to be solved. Every input is connected to every output and all connection weights are adjustable. The network in trained over the entire training set until no significant error reduction is achieved [5]. Network error is then computed by running the entire training set. If the error is less than some threshold, the algorithm terminates. Otherwise, a hidden node is added to the network with its inputs coming from all input nodes and all other hidden nodes. The outputs of the new hidden node are not yet connected. The training values are then run over the

network with the input weights to the new hidden node being adjusted according to a maximization function. The outputs of the new hidden node are now connected to the output nodes and the input weights frozen while the output weights are trained. This is done in the same manner as was used to train the original network. If a significant reduction in error is achieved, the network training terminates; otherwise, the process recycles.

A comparison of the segmentation results obtained by the FCM, AFCM, and the FFCC showed that one method did not provide the best segmentation results for all cases. The FCM and AFCM displayed the best results for segmenting normal images while the FFCC provided better segmentation of tumorous regions. The FFCC, however, did not provide better segmentation results away from the tumorous region. This seems to indicate that the operator needs to have an idea of the nature of the image in order to apply the 'best' segmentation algorithm [5].

An automated image interpretation technique is presented in [8] that uses genetic algorithms (GA) to perform image segmentation and labeling. A hypothesize-and-verify strategy is used in which image segmentation and labeling solutions are generated by the GA as hypotheses and then verified by an objective function. The 'fittest' hypotheses are reintroduced to the GA and new hypotheses are generated. This process repeats until until the GA converges to an optimal image interpretation.

The GA based method begins by oversegmenting an image with an edge-based region growing technique to construct homogeneous regions. Oversegmentation is preferred because further processing will only merge regions instead of splitting them and merging is retrospectively an easier process. A GA code string is then constructed with each position of the code string corresponding to a numbered region of the segmented image and the value of each position in the code string being the (hypothesized) region label. Using an initial population of code strings, new code strings are formed and their fitness is evaluated. Fitness is evaluated against an objective function which represents *a priori* knowledge of the brain anatomy and imaging parameters. Function results are based on image regions and the relationships between them.

2.3.4 Classification. Classification is the last step in the process of Figure 2.1. During segmentation, a pixel, based on features, is assigned to a particular class. However, some methods make no connection between the segmentation classes and the tissue classes. They simply group like pixels or regions together. Strictly speaking, this is truly segmentation.

Some methods incorporate the domain knowledge of the application into the segmentation step to arrive at both segmentation and classification simultaneously. The process developed by Tsai [1] combines general anatomical knowledge of the human brain with image processing techniques to actually obtain classification. The process uses a combination of pixel intensity analysis, morphological operations, and anatomical knowledge to classify brain tissues. This combination takes advantage of each of these components and uses two MR image types. As regions are identified, their locations and statistical information give knowledge used for further segmentation. This implies that while there are global rules and knowledge that guide the segmentation, information gleaned during segmentation allows adaptation during classification.

2.4 PCNN Theory

The PCNN is a physiologically motivated artificial neural network. The network model is based on stimulus-specific interactions between cells in primate primary visual cortex. Modulatory linking between cells has been observed in biology, therefore, this model was chosen because it simulates this inter-neuron linking.

Neurons have the characteristic or ability to respond to stimuli. This response is called *firing*. A neuron fires when its internal activity reaches a certain threshold. The internal activity of a neuron is determined by its inputs. For the PCNN, inputs are in the form of primary feeding inputs and secondary linking inputs.

A model PCNN neuron consists of inputs that model dendrites and a pulse generator. The dendrites receive one or more feeding inputs that are modulated by one or more linking inputs. Linking inputs are the lateral connections between neurons. The pulse generator compares the multiplicatively modulated feeding signal to a variable threshold to determine

firing sequence. A graphical representation of a model neuron is given in Figure 2.2. See [9] for the theoretical and biological foundations of the PCNN.

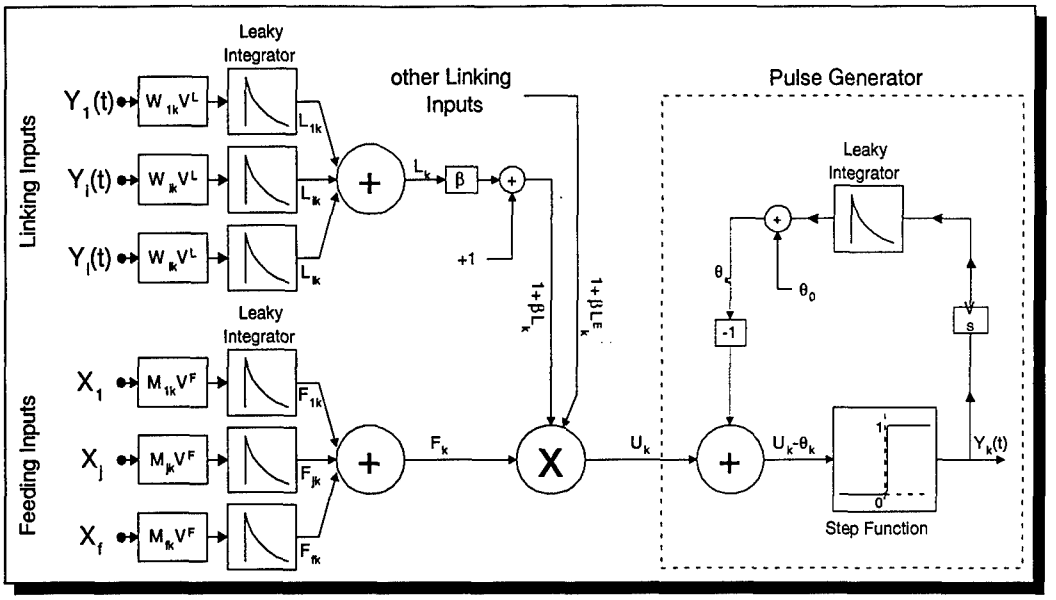


Figure 2.2 Model PCNN neuron

As stated, the internal activity U of neuron k at time t is given by

$$U_k(t) = F_k(t)[1 + L_k(t)], \tag{2.1}$$

where $F_k(t)$ is the feeding signal to neuron k and L_k is the linking signal [10]. The feeding and linking signals are each the result of a sum of a series of leaky integrators. The leaky integrators turn pulsed inputs into an exponentially decaying persistent output. A neuron, N_k, will fire when its internal activity, U_k, exceeds its variable threshold, θ_k. The threshold is variable in the sense that as soon as it is charged it begins to decay exponentially. The threshold becomes charged when a neuron fires. Therefore, according to the model, all neurons fire at time $t = 0$. Eventually, as θ_k decays and U_k increases, all neurons with $F_k > 0$ will fire. For mathematical descriptions of the model neuron see [9] and [10].

2.5 Conclusion

This chapter introduced the Magnetic Resonance image and discussed current segmentation methods in the context of a generic image analysis system. It also introduced the concept of the pulse coupled neural network. While there are many filtering and segmentation methods available, the PCNN's physiological foundation in the primate visual system may hold promise for an MR image segmentation method.

Certain characteristics of current segmentation approaches have shown value and will be incorporated in the method developed. These traits include spatial information, noise removal, and contrast enhancement. However, the PCNN method will also attempt to overcome limitations of current methods. Limitations include training, human supervision, and segmentation reproducibility. The next chapter will specifically describe the adaptation and application of the PCNN to perform both image filtering and segmenting.

III. Approach

3.1 Introduction

In the previous chapter, the formation of MR images was introduced. Current segmentation approaches were also discussed in the context of a generic image analysis system and how the nature of the MR image itself presents great difficulty to these methods. Finally, a new neural network was introduced. This chapter will describe image visualization and manipulation. The implementation of the PCNN as a physiologically motivated artificial neural network will be presented and demonstrated for segmenting and filtering.

3.2 Image Manipulation

This section briefly describes the processes and requirements for obtaining the MR brain images and viewing the images prior to and after image segmentation.

3.2.1 Image Acquisition. MR images used were obtained from a General Electric Signa MRI scanner. Images were originally obtained by the Computerized Anthropometric Research and Design Laboratory, Armstrong Laboratories, in cooperation with the MRI facility, Wright Patterson Medical Center. Each image is a $256x256$ array of 16 bit binary gray scale intensity values. A detailed description of the data transfer process used is given in Appendix A.

3.2.2 Image Viewing. Images extracted from the archives could now be viewed. Each original image contained a header with information regarding the patient, scanning facility, and scanning parameters. This header was removed prior to filtering and segmenting. Only pixel information is given to the PCNNs.

Actual pixel data begins immediately after the image header and is in row major order. Each pixel is 16 bits long and represents a gray scale intensity. Matlab was chosen for image manipulation because of its flexibility, image processing, and mathematical processing power. A description of the routines used for image viewing and manipulation can be found in Appendix B.

3.3 PCNN Based Segmentation Method

Images were segmented using a method developed based on the pulse coupled neural network. The PCNN theory was adapted and implemented to perform segmentation of 2D MR images. The two-dimensional PCNN was then adapted to work on three-dimensional volumes. The original PCNN took into account neighbors only in the planar sense. This is acceptable if dealing with truly two-dimensional data. However, the MRI data was extracted from a three-dimensional volume. Since the data comes from a volume represented by a series of slices, a particular slice has a relationship with those above and below it. This relationship is captured in the 3D implementation of the PCNN and allowed to influence image and volume segmentation.

The overall segmentation method used to segment MR brain images consists of three primary stages and is illustrated in Figure 3.1. First, an original image is PCNN filtered as described in Section 3.3.3. Next, the filtered image is contrast enhanced. Finally, the contrast enhanced image is PCNN segmented via the processes discussed in Sections 3.3.5 and 3.3.6 to arrive at the final segmentation.

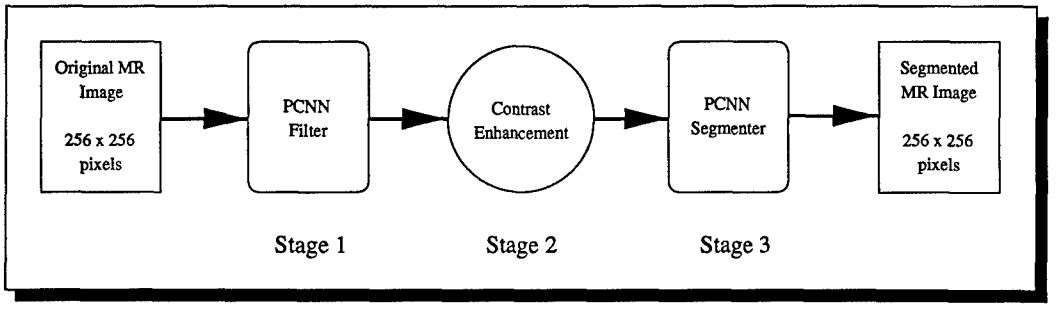


Figure 3.1 MRI segmentation method

3.3.1 Image Processing PCNN. The modulatory behavior of neurons is the distinguishing factor of this neural network model. This ability is what allows for image segmentation. Neurons with like inputs are able to fire at the same time because of the modulatory linking. Pixels similar in intensity are therefore able to link and pulse synchronously. The internal activity equation for an image processing neuron is

$$U_k = F_k(1 + \beta L_k), \tag{3.1}$$

where F_k is the pixel intensity fed to the k^{th} neuron, L_k is the total linking input to the neuron, and β is a bias that represents the value of the linking strength between neurons [10].

As applied to image segmentation, the PCNN's neurons receive feeding inputs from gray scale image pixels and linking inputs from neighboring neurons. The network architecture consists of one layer of neurons with each neuron corresponding to an image pixel. Each neuron is also connected to a number of neighboring neurons defined by a neighborhood linking radius. A graphical representation is given in Figure 3.2.

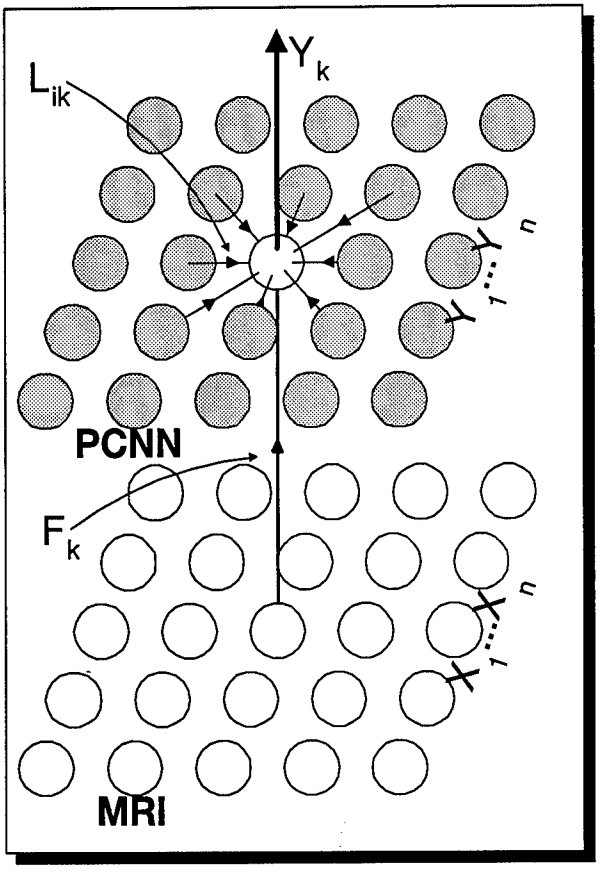


Figure 3.2 Neuron feeding and linking connections

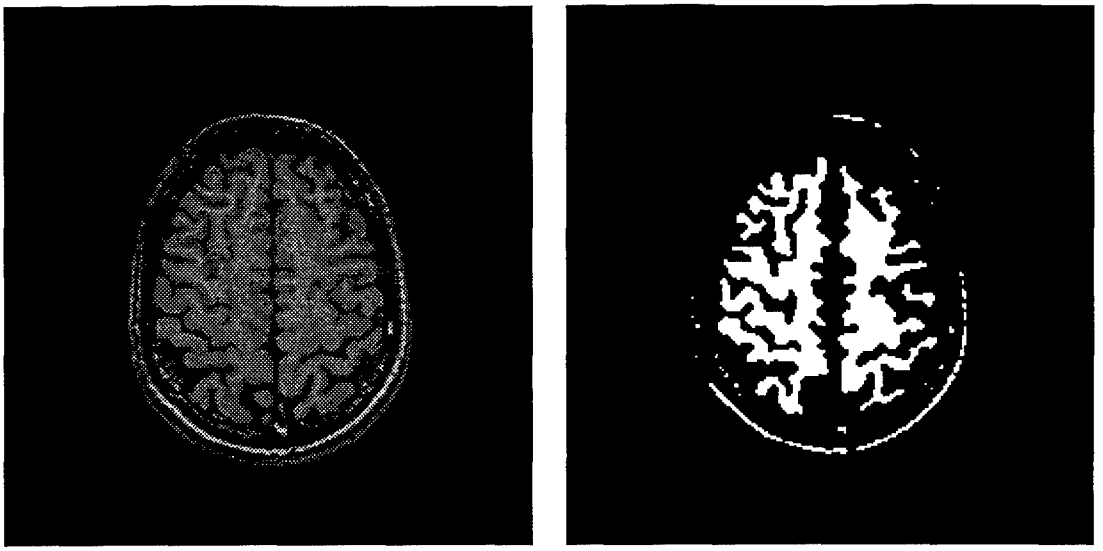
The PCNN segmentation algorithm proceeds in the following manner. Since all neurons are considered to have fired at time $t = 0$, the segmentation begins at time $t = 1$ with all θ_k fully charged. All F_k are normalized by the maximum intensity of the image so that at time $t = 1$ the neurons corresponding to the brightest pixels will fire.

Once a neuron fires, each of the linking fields, L_k, of the neurons in the firing neuron's neighborhood receives a pulse. If sufficient linking is achieved, *i.e.*, $\theta_k < F_k(1+\beta L_k)$, these neurons will fire also. Therefore, neurons may have to be evaluated many times before they fire or it is determined that they won't fire during a timestep. Once a determination has been made for all neurons during a timestep, a steady state is achieved. Time is then advanced, θ_k decays, and the process repeats until all neurons have fired or the given number of timesteps has been reached.

Here the image processing algorithm implemented departs from the PCNN theory. Instead of allowing each neuron to pulse throughout the course of the algorithm, only the timestep of the first pulse of each neuron after time $t = 0$ is recorded. This timestep value is indicative of the assigned class. Therefore, all neurons that pulse during a given timestep are assigned to the same class. A sample image and the results of the first few segmentation timesteps is shown in Figure 3.3.

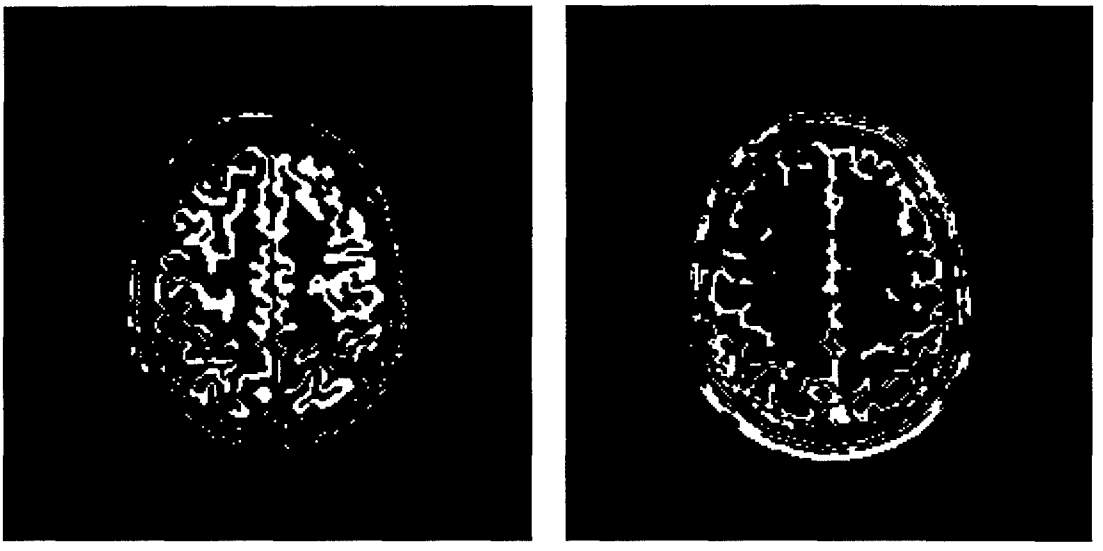
Another difference between theory and implementation is that linking fields are only valid for the timestep for which they were generated. The idea behind this being as follows. The linking field is providing a modulatory influence on a neuron. If neurons are similar but not identical, the linking field may be strong enough to bring out the similarity and cause the neurons to pulse together and therefore be classified together. However, if a neuron is allowed to influence another neuron to fire at a later timestep, the neurons will not be classified together even though one was influencing the other.

A modification to the manner in which the neuron's linking field is calculated allows the PCNN to operate more efficiently in software. The modification is best demonstrated by examining when the linking field is updated. A neuron's linking field is only useful when surrounding neuron's have pulsed. Therefore, one software implementation of the hardware model neuron of Figure 2.2 would be to calculate a neuron's linking field based on the current state of surrounding neurons *every* time a neuron's internal activity is calculated. It can be seen that this approach is inefficient. If one neuron in a neighborhood were to fire, then every neuron's linking field in that neighborhood would need to be recalculated to see if neighboring neurons could now fire. Because a neuron's linking field is a sum of



(a) Original MR Image

(b) Timestep 1



(c) Timestep 3

(d) Timestep 5

Figure 3.3 Timestep segmentation results

weighted inputs, a linking radius of 1 results in 8 neighbors updating their linking fields, or 128 calculations.

Another approach is to update neighboring linking fields when a neuron fires. By examining Figure 2.2, it can be seen that when a neighboring neuron fires, the result is a simple weighted addition to the current value of the linking field. Since the values of the weights and the magnitude of the linking pulse are known, the weighted addition can be calculated prior to starting the PCNN. As a result, when a neuron fires, simple additions can be performed on the neighboring neurons' linking fields. For a linking radius of 1, only 8 additions are performed. This concept is demonstrated in Figure 3.4.

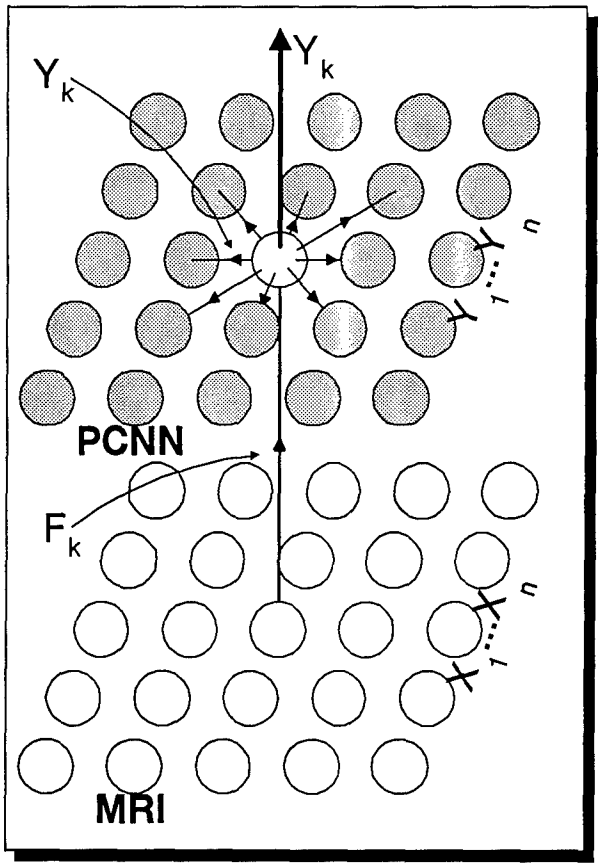


Figure 3.4 Revised neuron feeding and linking connections

Conceptually, Figure 3.2 and Figure 3.4 are the same. The first represents the fact that linking inputs are received or gathered from neighboring neurons, and the second represents linking inputs as being sent out to neighboring neurons. In both cases, linking inputs are being provided to neighboring neurons. The difference is in the representation and calculation of the linking field's value. The approach taken here uses the latter.

Another departure from PCNN theory is the implementation of the decaying threshold, U_k. Theoretically, each neuron's pulse generator contains its own threshold because neurons were allowed to pulse multiple times and at differing rates, therefore, with different thresholds. However, since neurons are only allowed to pulse once, and every neuron's threshold begins at the same value and decays at the same rate, only one threshold value is required.

A revised version of the model neuron of Figure 2.2 is given in Figure 3.5.

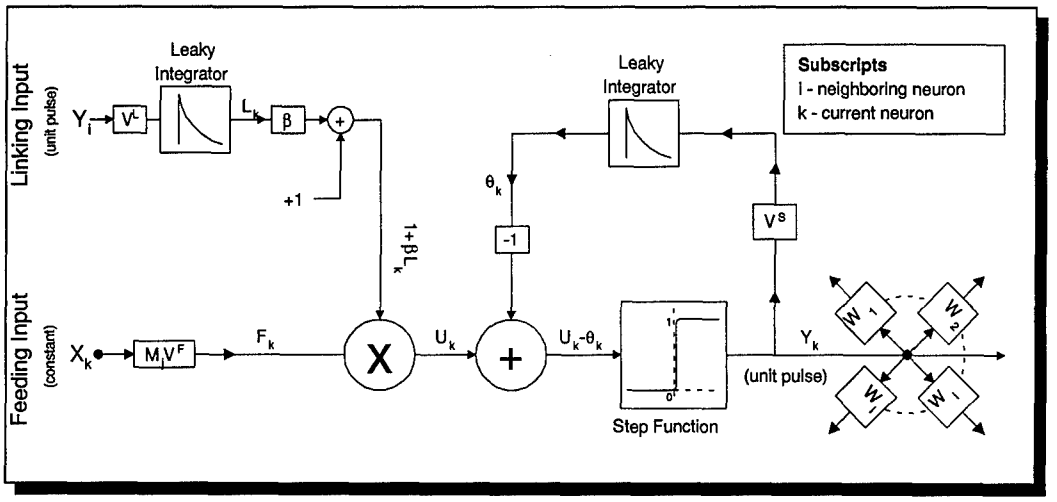
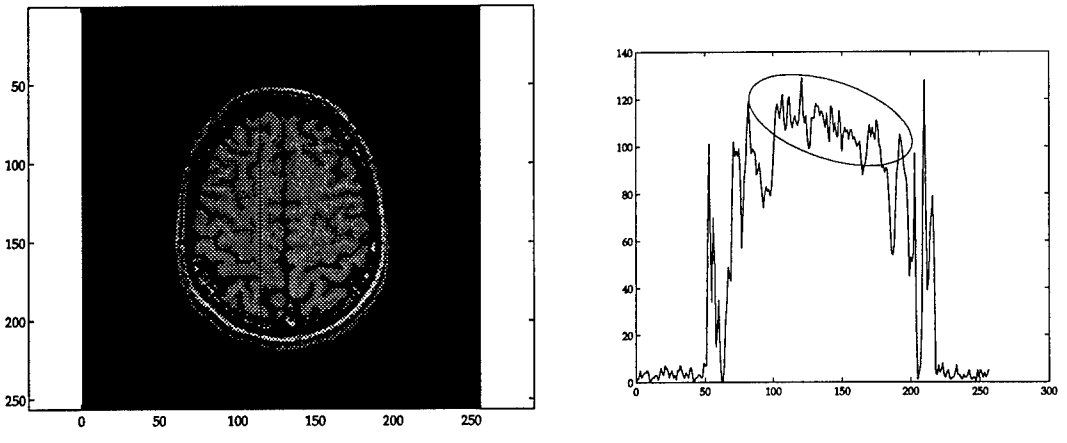


Figure 3.5 Revised model neuron

3.3.2 Original Images. Figure 3.6(a) and 3.6(b) represent typical MR brain images used in this research. These images are unfiltered and unsegmented. Image noise and spatial intensity variations are apparent. To further illustrate the existing variance, a plot of intensity value versus pixel location for the vertical reference line drawn in Figure 3.6(c) is represented in Figure 3.6(d). Note the intensity variance in the circled region of Figure 3.6(d). This is a high degree of variance for what appears to be pixels of the same region.

3.3.3 PCNN Filter. Images first had to be filtered prior to segmentation. This was done to reduce the local spatial variance that existed within each image, reduce the

(a) (b)



(c) (d)

Figure 3.6 Typical MR brain images are represented by (a) and (b) with (d) representing the intensity plot of the reference line in (c).

number of pixel intensities in the image, and eliminate noise artifacts present from the scan.

The PCNN was also used to perform image preprocessing. A weakly linked PCNN was fed the original input images and allowed to run. After each epoch, the firing order of the neurons representing the pixels were examined and their corresponding intensities adjusted. Figures 3.7, 3.8, 3.9, 3.10, and 3.11 illustrate the progression of an original image through the filtering process. PCNN filtering is a gradual process and is performed over multiple epochs. Changes from one image to the next are not always apparent, however, an appreciation for the overall result can be achieved by comparing the original image to the final. The reason changes aren't apparent from one stage to the next is that *each* epoch results in a number of pixel intensity adjustments. The decision to adjust a pixel's intensity is based on neighboring neurons.

As stated earlier the neuron's neighborhood is defined by its linking radius. If a majority of a neuron's neighbors fired before the specified neuron, that neuron's corresponding pixel intensity was increased. Since the neuron's feeding input is the pixel intensity, an increase in the feeding value should increase the neuron's internal activity and cause it to fire sooner. Likewise, if a majority fired after, the intensity was decreased. If a majority fired at the same time, no action was taken. Once the number of intensity adjustments dropped below a certain threshold, filtering stopped. The threshold allows the monitoring of the number of pixels whose intensity values change during an epoch.

Figure 3.12 illustrate the results of applying the PCNN based filter to the original images in Figures 3.6(a) and 3.6(b). Figures 3.6(a) and 3.6(b) originally contained 164 and 219 different intensity values respectively. After PCNN filtering, the number of intensities were correspondingly reduced to 74 and 95 intensities. Both images used parameter settings of $\tau_S = 103$, $T = 63$, $V_S = 1.0$, $\beta = 0.01$, and a filtering threshold of ten percent.

The resulting images demonstrate a clear reduction in the variance and number of pixel intensities. However, there is also an obvious reduction in image contrast.

3.3.4 Contrast Enhancement. One of the effects of filtering is the reduction in image contrast. Since the PCNN is physiologically based on the primate visual system, an improvement in contrast should aid segmentation. Therefore, a histogram equalization procedure is used to perform image contrast enhancement. Figures 3.13(a) and 3.13(b)

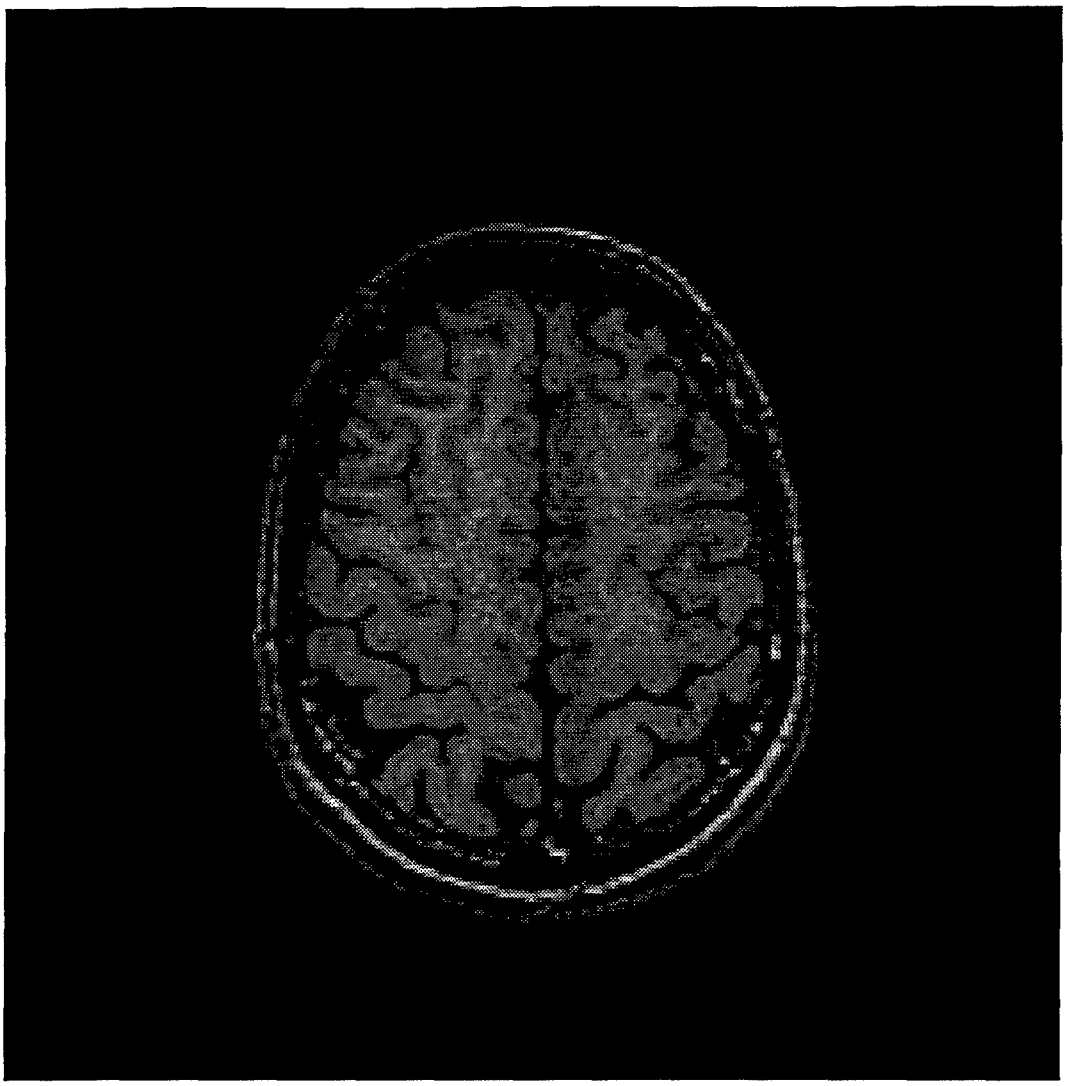


Figure 3.7 Original MR image

demonstrate the results of applying histogram equalization to the PCNN filtered images to improve contrast. Comparing these figures to those of the filtered images illustrates the effectiveness of the enhancement. The low contrast of the filtered images makes it difficult to discern intensity differences, while the enhanced images make intensity differentiation easier. Additionally, outliers not handled by the filtering process are dealt with.

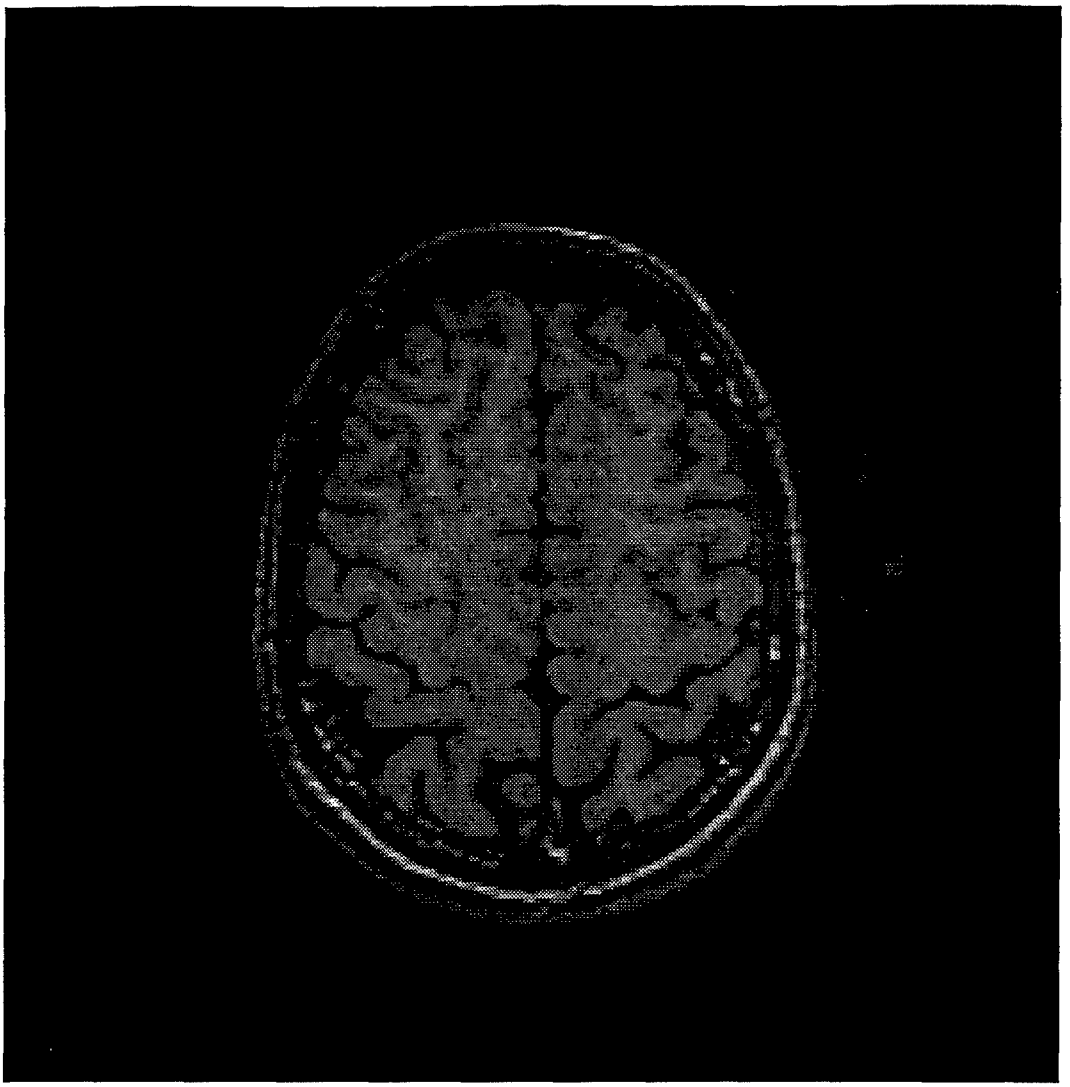


Figure 3.8 Results after 1 epoch

3.3.5 2D PCNN Segmenter. The actual PCNN segmentation proceeds much the same way as the filter process with the exception of the intensity adjustment and strength of the bias field. Intensities are now fixed and used as features. Since it is assumed that filtering enhanced the relationships between pixels, the linking field bias is increased. A filtered and contrast enhanced image is presented to the PCNN and the image is segmented.

Figures 3.14(a) and 3.14(b) illustrate the results of applying the PCNN segmentation algorithm to the PCNN filtered and contrast enhanced images. Parameter settings of

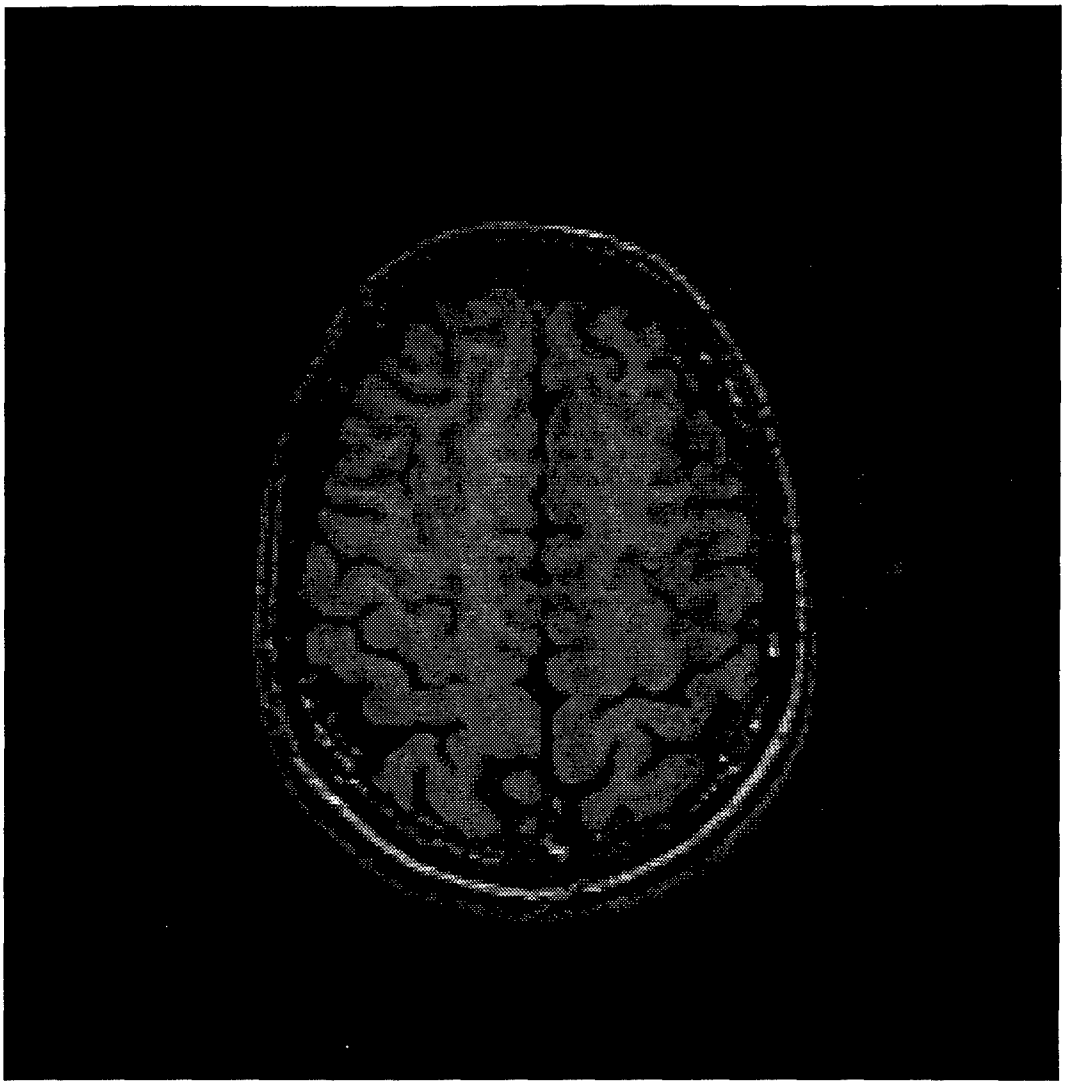


Figure 3.9 Results after 2 epochs

$\tau_S = 25$, $T = 14$, $V_S = 1.0$, and $\beta = 0.1$ were used to obtain both images. The segmented images now contain only 9 different intensity values. It can be seen that the segmentation has grouped pixels and regions together that originally suffered from a great deal of noise and variance. Image detail and object boundaries are generally maintained.

3.3.6 3D PCNN Segmenter. There are two forms of three dimensional segmentation in this research. The first uses the slices directly above and below the slice to

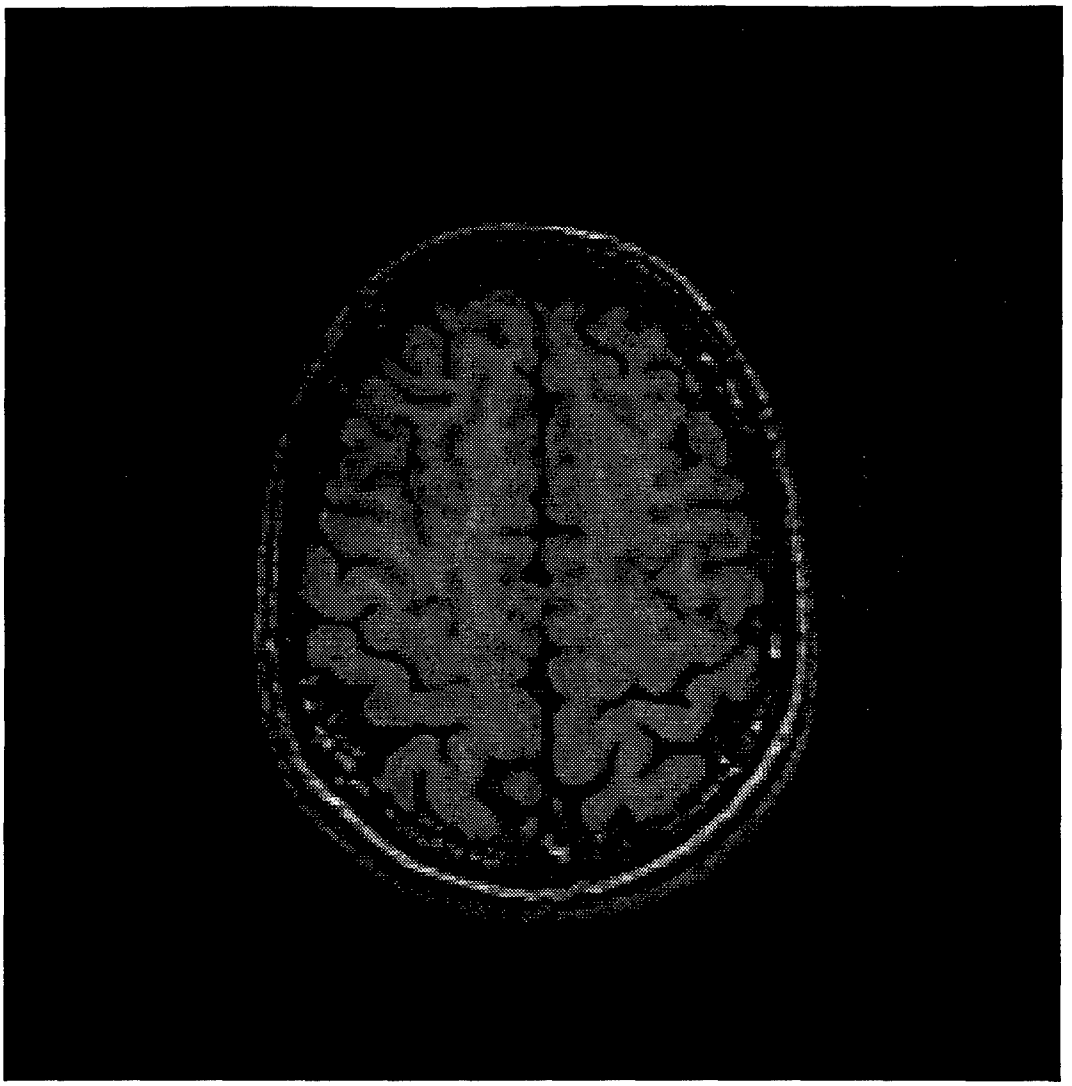


Figure 3.10 Results after 3 epochs

be segmented as additional information for the linking field of the PCNN. The second approach actually segments an entire volume represented by a series of images.

Figures 3.15, 3.16, 3.17, and 3.18 are representative of a series of 2D images used to create and segment a 3D volume. The process used for this procedure was the same as the process used for the 2D images. The parameters for 3D PCNN filtering and segmenting are also the same as the 2D parameters.

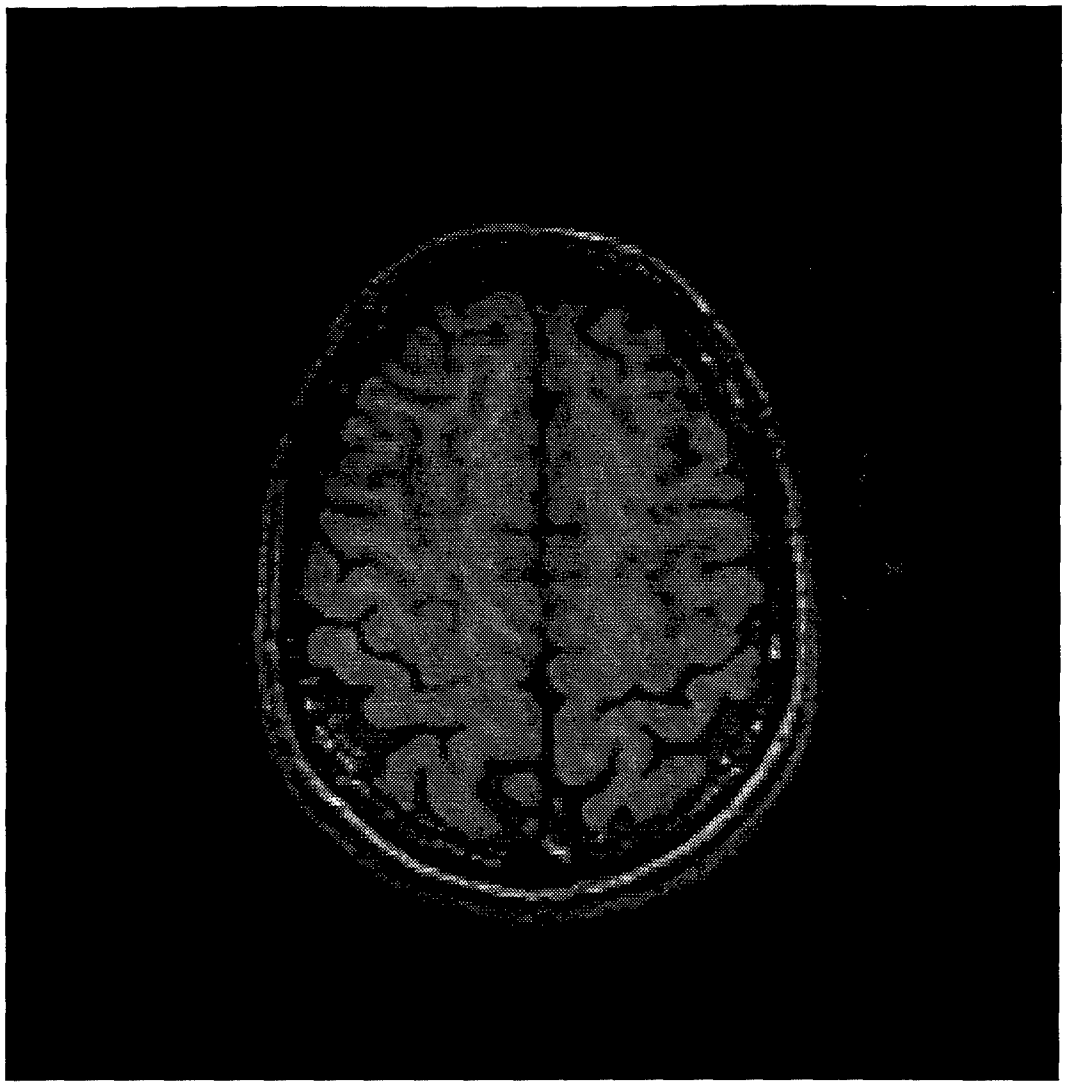
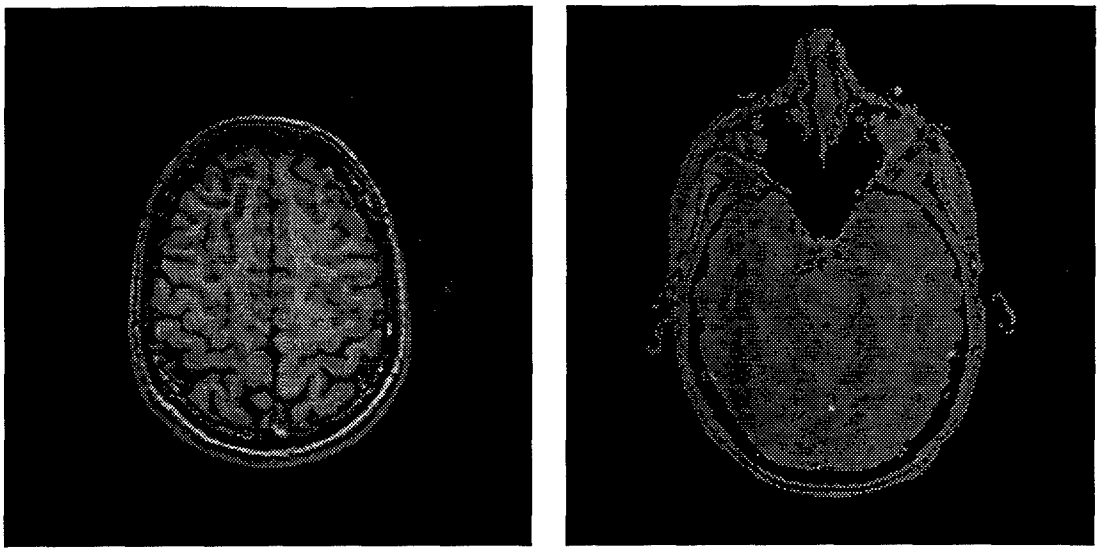


Figure 3.11 Results after 4 epochs

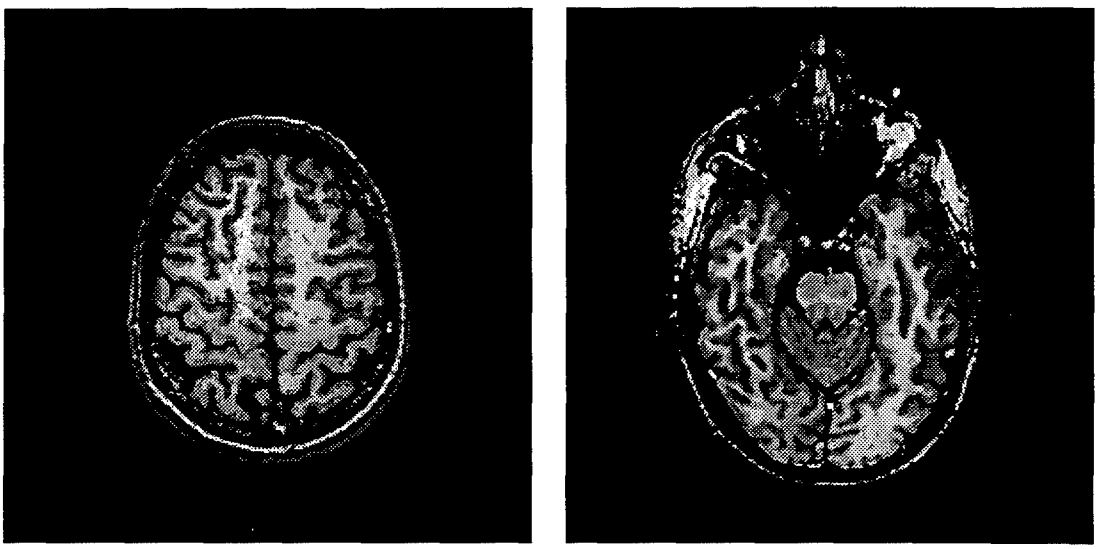
Figure 3.15 is a subset of a set of continuous slices used to construct a volume to be segmented. Structure and intensity similarity can be observed from image to image. Only small changes occur from one image to the next.

Figure 3.16 is the result of applying the 3D PCNN filter to the images of Figure 3.15. As with the 2D PCNN filter, intensity variance is reduced at the expense of image contrast.



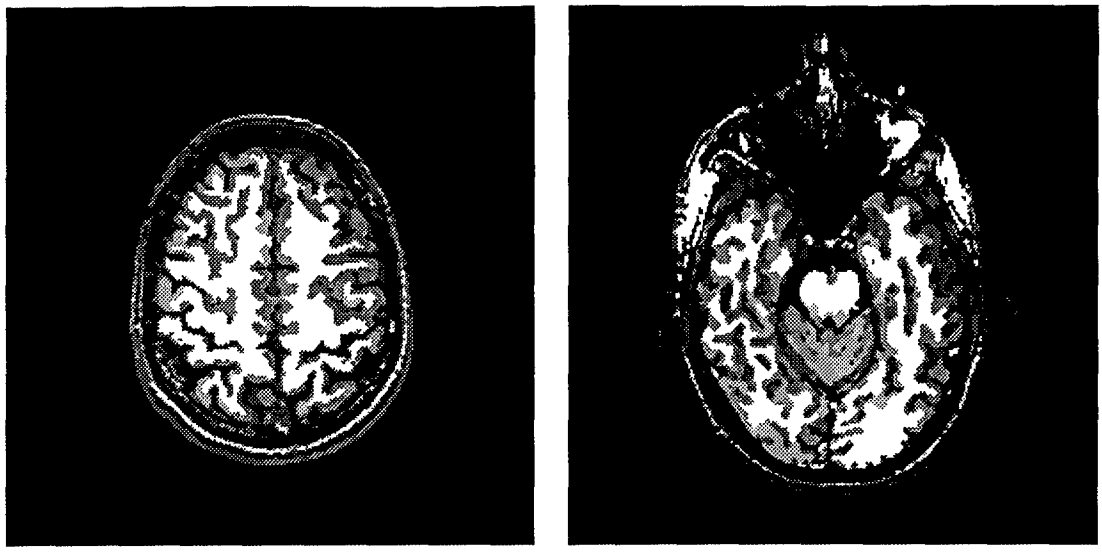
(a) (b)

Figure 3.12 PCNN filtered images



(a) (b)

Figure 3.13 Filtered images after histogram equalization



(a) (b)

Figure 3.14 PCNN segmented images

Figure 3.17 is the result of applying contrast enhancement to each of the images in Figure 3.16. Again, like the 2D approach, contrast is improved and structure and intensities are easily differentiated.

Figure 3.18 is the result of applying the 3D PCNN segmenter to the filtered and contrast enhanced volume. The segmentation has again grouped like pixels together while maintaining more details and boundaries than the 2D method.

Figure 3.19 was obtained using the 3D filter and segmentation processes illustrated previously with a slight modification. The image was originally segmented using the 2D process (Figure 3.14(b)) and then segmented using the 3D process except only the images directly above and below were used to construct the volume. The same parameters were used for both segmentations. The differences are quite noticeable. An examination of the boxed area of Figure 3.19 shows details not captured by the 2D segmentation method.

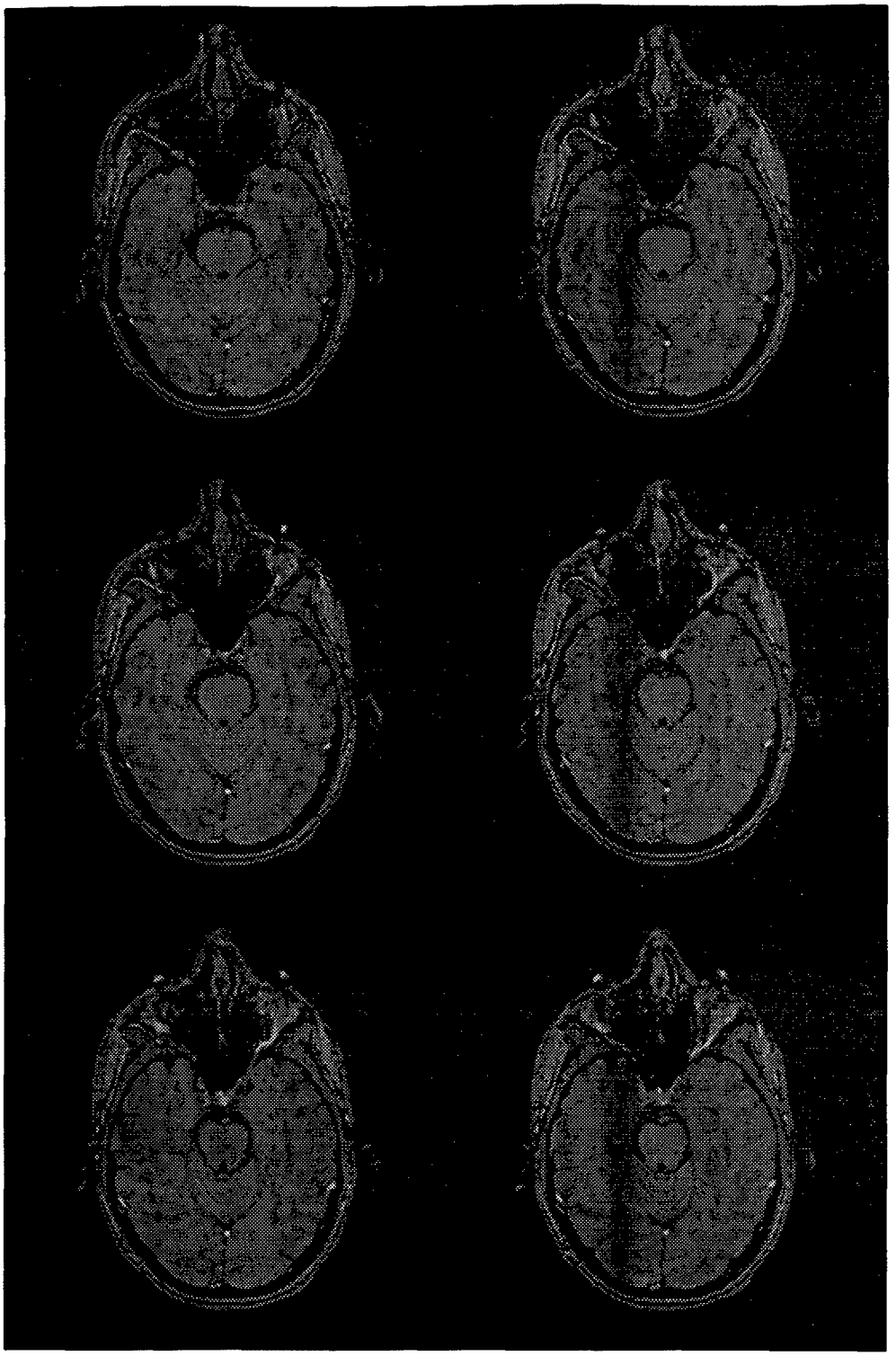


Figure 3.15 Montage of six original MR brain images used for volume construction

Figure 3.16 Montage of filtered version of images of Figure 3.15

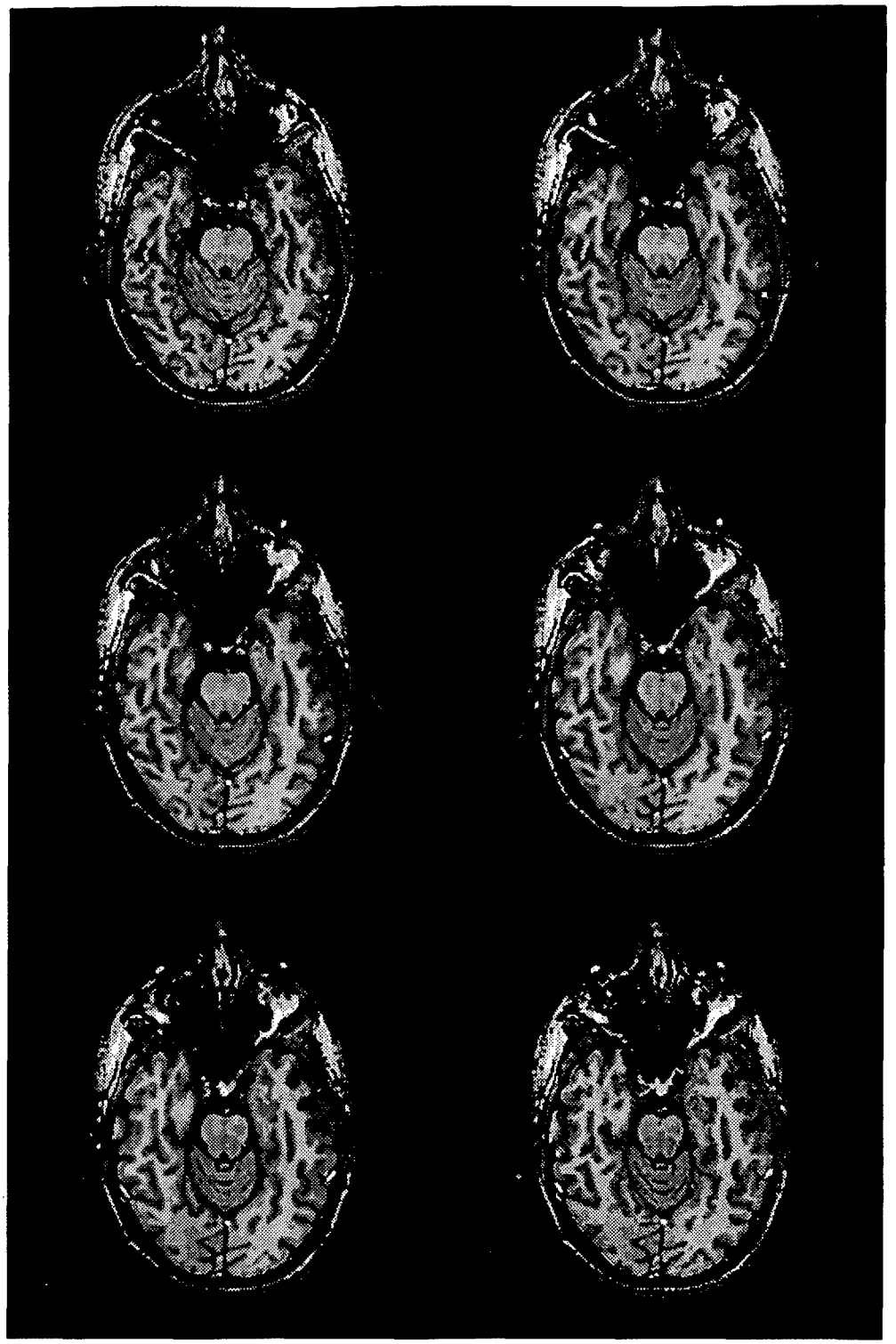


Figure 3.17 Montage of images of Figure 3.16 after contrast enhancement

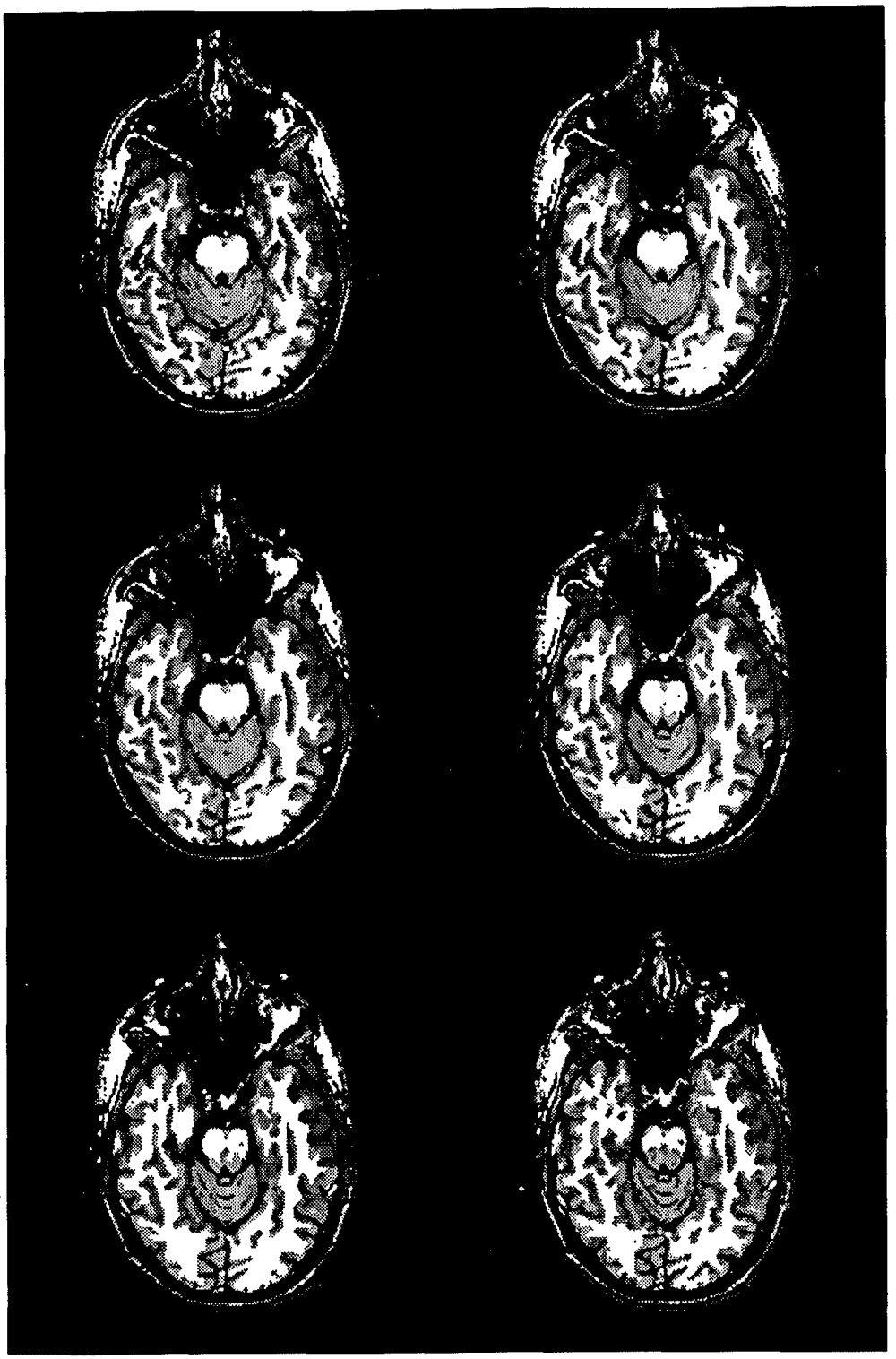


Figure 3.18 Montage of final segmented images

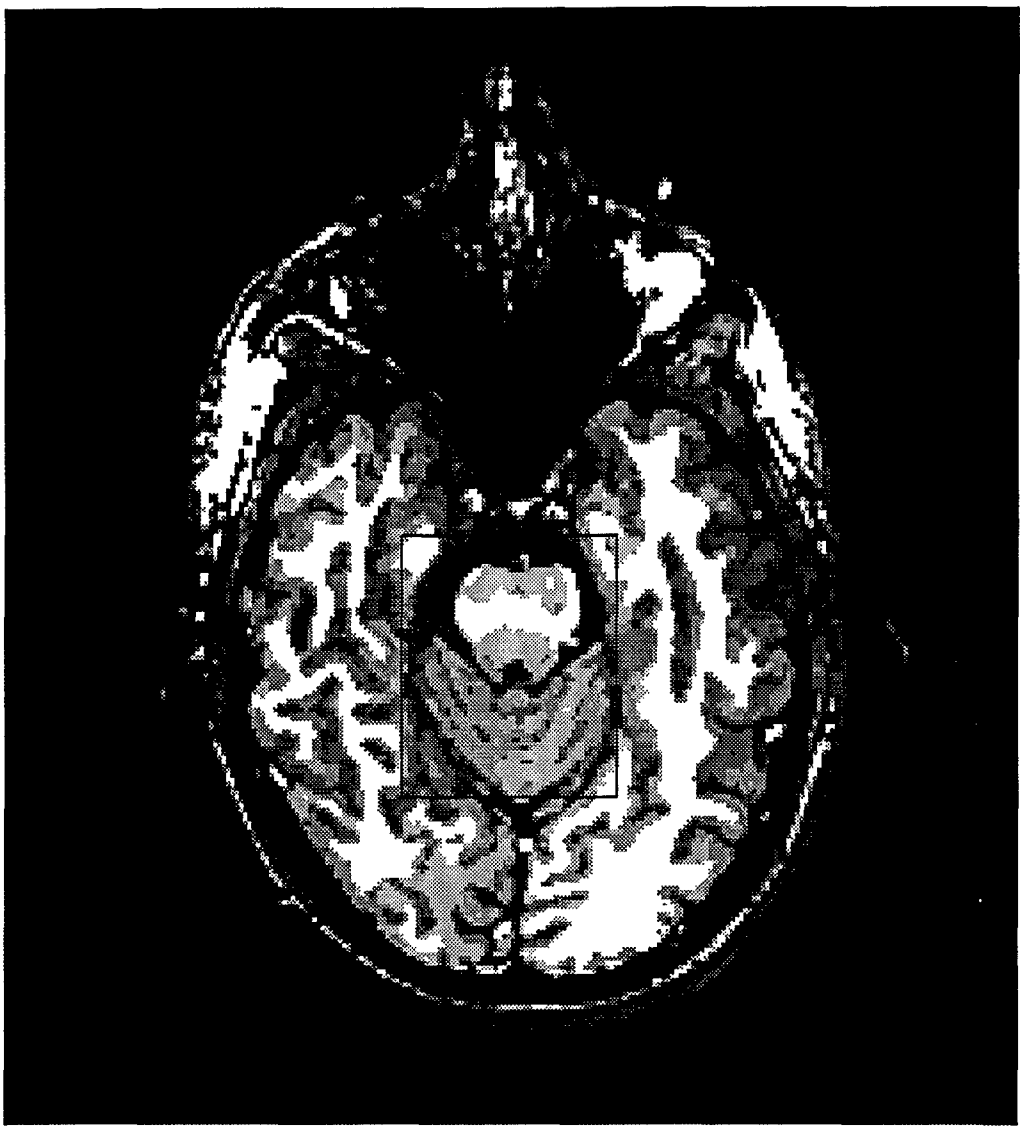


Figure 3.19 3D segmentation of image in Figure 3.6(b)

3.4 Segmentation Analysis

Visually, the results of the processes described and illustrated thus far *appear* to produce the desired results. However, visual examination of the resulting images does not validate the effectiveness of the method. A comparative analysis of the resulting regions' intensities to corresponding original image intensities should give an indication of the performance of the method. Figure 3.20 shows a mapping of the original image pixel intensities of Figure 3.6(a) to the new image pixel intensities of Figure 3.14(a). The number of bars in the figure represent the number of resulting intensities in the segmented image. It can be seen that the PCNN has grouped pixels together whose intensities vary widely.

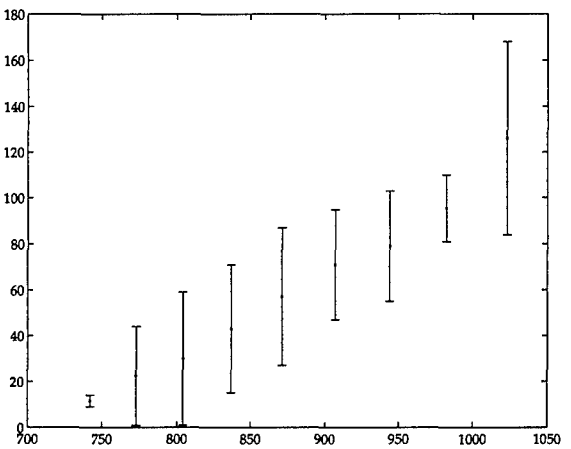


Figure 3.20 Segmented intensity to original intensity mapping

3.5 Software Architecture

An object-oriented approach following Rumbaugh [11] was taken to develop and implement the PCNN based segmentation method. This was done to assure ease of growth and adaptability. The object-oriented design and implementation allows for the application to varying data sets and purposes, not just MR image segmentation. A more detailed description of the software architecture, its implementation, and its use can be found in Appendix C.

3.6 Conclusion

This chapter presented the implementation and results of the PCNN as a software system to perform image segmenting and filtering. Changes to the hardware model neuron were defined and implemented to allow the PCNN to function more efficiently and to operate on multidimensional data. The next chapter presents conclusions and recommendations for future research.

IV. Conclusions

These conclusions and recommendations are based upon the results detailed in the previous chapters.

4.1 PCNN Segmentation Method

A segmentation method based on a model of the pulse coupled neural network was implemented and tested on MR brain images. Certain adaptations were necessary in order to use the PCNN as an image analysis tool. These adaptations include the modification of the neuron's feeding field, the single pulse neuron, single timestep linking, and a single threshold decay variable used by all neurons.

These modifications to the model neuron resulted in over an order of magnitude performance increase as well as reducing the memory requirements by over thirty percent. These results allowed the extension of the PCNN to operate three dimensionally. This new application of the PCNN results in a better segmentation of the image or volume of interest. This is primarily due to the additional spatial information that the PCNN is capable of incorporating.

A mapping of each segmented region's new pixel intensities to original image intensities demonstrates that the PCNN segmentation method developed by this research performs more than just simple thresholding. This research clearly demonstrates that the PCNN can be used as an effective image analysis tool.

4.2 Recommendations for Further Research

There are several key areas that could be examined by further research. These include the implementation of a true PCNN feeding field, incorporation of pulse inhibition, extending the method to perform tissue classification rather than segmentation, and an in depth examination of the effects of PCNN parameter settings and training.

Appendix A. Image Extraction

This appendix provides the processes to extract archived MR brain images from 4mm DAT tape.

A.1 Image Extraction Procedures

Once the data was acquired from the scanner, it was archived on 4mm DAT tapes and 5 1/4" Optical Disks. However, the data was only readily accessible from a GE MRI or CT scanning console. This is because the data was written to the tapes and optical disks in a GE format. The images were also compressed by a GE compression routine prior to archiving. In order to extract the data from these mediums, two different processes were used. One process required access to a General Electric (GE) MRI or CT console. Another process only required equipment capable of reading 4mm DAT tapes.

A.2 Method One

The first process required access to a GE console. The desk where the technician controls the MRI or CT scanner and equipment is referred to as the console. The GE CT console at the Wright Patterson Medical Center was used. In order to extract archived images, they first needed to be restored to the console's hard drive. Once restored, they resided on a GE partition of the system's hard disk. Restoration was performed by the CT technician. Next, the images needed to be decompressed and placed in a unix partition of the hard disk. To accomplish this task, access to the operating system was required. A command shell started from the utilities menu provided access. The following steps extract and save restored images.

Step 1 Restore Archived Data

1. Place 4mm tape with archived data into the drive.
2. Have a technician restore data from the console.
3. Obtain exam, series, and image numbers from technician.
4. Once data is restored, remove tape from drive.

Step 2 Access Unix

1. At the console, touch "Utilities"
2. Touch "Shell"

Step 3 Extract Images

1. From the command shell, type: cd /usr/g/insite/bin <enter>

   This is the location of the image extraction utility.

2. To extract images, type: ximg -i e(exam)s(series)iall -s -t <enter>

   This command will extract all the images from the given exam and series without stripping patient information and will save images in rectangular, decompressed form. Images are located in the /usr/g/insite/tmp directory. Use the exam and series information obtained from restoration step. If all the restored data must be extracted, reissue the above command with the appropriate exam and series numbers until all desired data is extracted.

Step 4 Save Extracted Images to Tape

1. Change directories. Type: cd /usr/g/insite/tmp <enter>
2. Place a new tape in the 4mm tape drive.
3. To save images to tape, type: tar -cvf /dev/nrst8 E* <enter>

Step 5 Remove Extracted Images from Console

1. To remove, type: rm E* <enter>
2. To verify, type: ls <enter>

Step 6 Close Command Shell

Step 7 Remove tape from 4mm tape drive.

Images now reside on the new tape in a format readable by any machine capable of reading 4mm tapes and executing the tar utility.

A.3 Method Two

The second process used to restore images from an archived tape used a public domain utility called dicom3tools. Dicom3tools contains many routines for converting images to the new Dicom standard. One of the routines has the ability to read GE MRI archived data from an external device. Once the images were restored, they needed to be decompressed. A decompression routine was implemented using the GE pseudocode decompression routine. The following steps illustrate the process.

Step 1 Install Software

1. Install and compile Dicom3tools on the system with the external device that will be used to read the data.
2. Install and compile the GE decompression routine.

Step 2 Read Data

1. Change to the directory where you wish the data to reside.
2. Place medium with archived data in device.
3. To remove data, type: ex9t (device name) <enter>

Step 3 Decompress Data

To decompress an image, type: decompress (source) (target) <enter>

Issue this command for every image file to be decompressed.

A.4 Decompression Code

```
#include "ge_structs.h"

main(int argc, char *argv[])
{
FILE                    *infile;
FILE                    *outfile;
char                    byte;
int                     i;
unsigned short int      real_pixels[256*256];
unsigned short int      base;
unsigned short int      first_byte;
unsigned short int      second_byte;
```

```
if (argc < 3)
  {
    printf("usage:  decom infile outfile\n");
    exit(1);
  }

if ((infile = fopen(argv[1], "r")) == NULL)
  {
    printf("input file could not be opened\n");
    exit(1);
  }

if ((outfile = fopen(argv[2], "w")) == NULL)
  {
    printf("output file could not be created\n");
    exit(1);
  }

fseek(infile, (2260+3180), 0);

base = 0;
for (i=0;i<256*256;i++)
{
  first_byte = fgetc(infile);
  if (first_byte < 0x40)
    base = base + first_byte;
  else if (first_byte < 0x80)
    base = base + (short)(first_byte - 0x80);
  else
    if (first_byte < 0xa0) {
      second_byte = fgetc(infile);
      base = base + (((short)(first_byte & 0x1f))<<8) + second_byte; }
    else if (first_byte < 0xc0){
      second_byte = fgetc(infile);
      base = base + (((short)(first_byte | 0xc0))<<8) + second_byte;}
    else{
      base = fgetc(infile);
      base = base << 8;
      base = base | fgetc(infile);}

  real_pixels[i] = base;
  fwrite(&base, sizeof(unsigned short int), 1, outfile);
}

}
```

Appendix B. Visualization

This appendix provides the functions used to view MR brain images represented by 16 bit binary values.

B.1 Image Viewing

```
function brain = getmribrain(filename, header_size, image_size1, image_size2)
% GETMRIBRAIN Load a 16 bit binary representation image
% brainimg = getmribrain(filename, header_size, rows, columns)
% by: Lt Shane L. Abrahamson

fid = fopen(filename, 'r');
% read header
if header_size ~= 0
  temp = fread(fid,[1,header_size], 'char');
end

% read image data
[brain, cnt] = fread(fid,[image_size1, image_size2], 'ushort');
brain = brain';
fclose(fid);

% display image
figure;
imagesc(brain)
colormap(gray)
axis('equal')
title(filename);
return
```

Appendix C. Software Architecture

This appendix provides a description of the software developed for the PCNN and how it is used.

C.1 Object Model

Figure C.1 contains the object model of the PCNN software developed. An MRI PCNN segmentation system consists of one or more PCNNs, a Reader, and a Writer. The PCNNs can be filters, segmenters, or pulsers. The PCNNs in turn consist of 3D arrays and a Stack. The design was implemented using the C++ programming language.

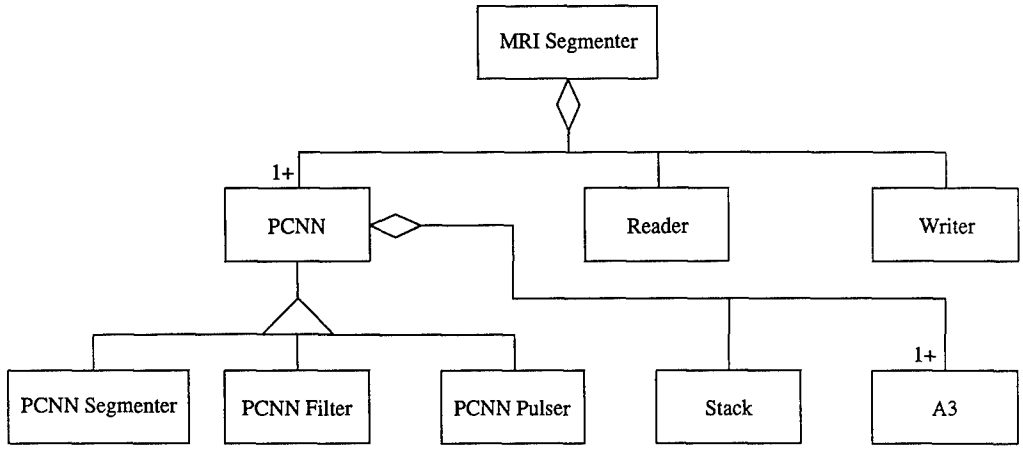


Figure C.1 PCNN Segmentation System Architecture

Conceptually, the various objects can be thought of as a set of filters through which the image data flows and is transformed. Figure C.2 demonstrates a type of program that could be constructed. In this instance, an image file is transformed by the Reader into an object that the PCNN filter can understand. The PCNN filter in turn, transforms the data into a filtered version of the original. The PCNN segmenter accepts the filtered data and segments before the Writer transforms it back to an image file.

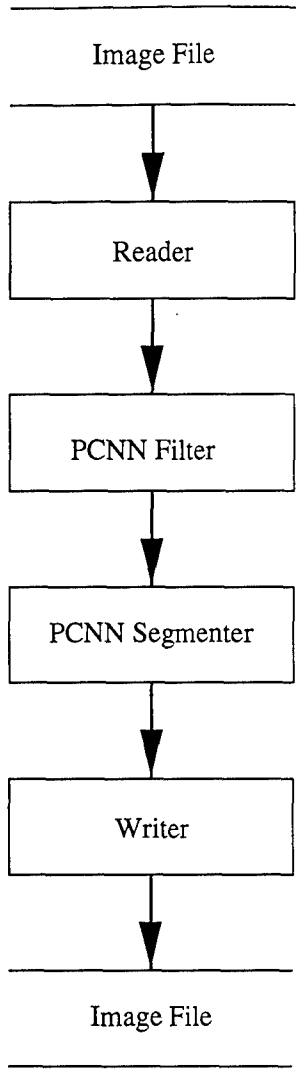


Figure C.2 Sample PCNN Object Flow Diagram

Each of the objects in Figure C.2 must be instantiated and its parameters set. A listing of each object's required parameters is given in the header of each object's source file. The main program in Section C.2 implements the concept of Figure C.2.

C.2 Sample Main

```
#include "PCNN_Segmenter.cc"
#include "PCNN_Filter.cc"
#include "Reader.cc"
#include "Writer.cc"
```

```
#include <fstream.h>

int main()
{
  PCNN_Filter    *my_filter    = new PCNN_Filter;
  PCNN_Segmenter *my_segmenter = new PCNN_Segmenter;
  Reader         *my_reader    = new Reader;
  Writer         *seg_writer   = new Writer;

  my_reader->SetDataDimensions(256, 256, 1);
  my_reader->SetFilePrefix("original_image");
  my_reader->SetImageRange(1, 1);

  my_filter->SetBeta(.01);
  my_filter->SetLinkingRadius(1);
  my_filter->SetLinkingTimeConstant(0.0);
  my_filter->SetThetaTimeConstant(100);//82
  my_filter->SetTimeSteps(60);//50
  my_filter->SetThetaV(1.0);
  my_filter->SetLinkingV(1.0);
  my_filter->SetInput(my_reader->GetOutput());
  my_filter->SetFilterThreshPercent(.1);
  my_filter->SetOutput((OutputType) 1);

  my_segmenter->SetBeta(.1);
  my_segmenter->SetThetaTimeConstant(25);
  my_segmenter->SetTimeSteps(15);
  my_segmenter->SetOutput((OutputType) 1);
  my_segmenter->SetLinkingRadius(1);
  my_segmenter->SetLinkingTimeConstant(0.0);
  my_segmenter->SetThetaV(1.0);
  my_segmenter->SetLinkingV(1.0);
  my_segmenter->SetInput(my_filter->GetOutput());

  seg_writer->SetFilePrefix("segmented_image");
  seg_writer->SetInput(my_segmenter->GetOutput());
  seg_writer->Execute();
  }
  return 1;
}
```

Appendix D. PCNN Code

This appendix provides source code listings of the C++ objects that implement a Pulse Coupled Neural Network.

D.1 PCNN Base Class

```
/*************************************************************
*
* Class PCNN4 contains the variables and methods necessary to
* implement a pulse coupled neural network.
*
* The following parameters MUST be set prior to use:
*   output
*   Timesteps
*   LinkingRadius
*   LinkingV
*   ThetaV
*   Beta
*   ThetaTimeConstant
*   input
*
/*************************************************************

#ifndef _PCNN4_
#define _PCNN4_

#include "NewStack.cc"
#include "A3.cc"

typedef struct neuron{
  int dep;
  int row;
  int col;
};

typedef enum OutputType {Timestep, Intensity};

class PCNN{

protected:

  A3<float>       *input;
  A3<float>       *LinkingField;
  A3<float>       *Theta;
  float           Theta1;
  A3<float>       *linking_weights;
  A3<float>       *results;
  Stack<neuron>   *PCNNStack;
  int             LinkingRadius;
  float           LinkingTimeConstant;
  float           ThetaTimeConstant;
```

```
  float          ThetaV;
  float          Theta0;
  float          LinkingV;
  float          Beta;
  float          PulseHeight;
  int            Redraw;
  int            TimeSteps;
  int            Timer;
  int            NumClasses;
  OutputType     output;

public:

//***************************************************************************
//
// Constructor initializes all variables of the pcnn to default values
//
//***************************************************************************

   PCNN()
     : input(NULL),
       LinkingField(NULL),
       Theta(NULL),
       Theta1(0.0),
       linking_weights(NULL),
       results(NULL),
       PCNNStack(NULL),
       LinkingRadius(0),
       LinkingTimeConstant(0.0),
       ThetaTimeConstant(0.0),
       ThetaV(0.0),
       Theta0(0.0),
       LinkingV(0.0),
       Beta(0.0),
       PulseHeight(MAXFLOAT / 100.0),
       Redraw(0),
       TimeSteps(0),
       Timer(0),
       NumClasses(0),
       output(Intensity)
  {}

//***************************************************************************
//
// SetInput assigns the pointer of the input class
//
//***************************************************************************
  void SetInput(A3<float> *PCNNInput)
  {
    input = PCNNInput;
  }

//***************************************************************************
//
// SetOutput assigns the type of output to produce
//
```

```
//*************************************************************************
  void SetOutput(OutputType value)
  {
    output = value;
  }

//*************************************************************************
//
// InitializeLinkingField instantiates a new instance of A3.  The linking
// field contains values used to calculate the contribution of
// neurons.  The array is padded and initialized to the value passed in.
//
//*************************************************************************
  void InitializeLinkingField(int rows, int cols, int depth, float value)
  {
    LinkingField = new A3<float>(rows, cols, depth);
    LinkingField->SetData(value);
  }

//*************************************************************************
//
// InitializeTheta instantiates a new instance of A3.  The Theta array
// contains the theta values of each neuron in the net and is used in the
// threshold comparison.  The array is also padded and initialized to the
// value passed in.
//
//*************************************************************************
  void InitializeTheta(int rows, int cols, int depth, float value)
  {
    Theta = new A3<float>(rows, cols, depth);
    Theta->SetData(value);
  }

//*************************************************************************
//
// InitializeWeights instantiates a new instance of A3.  The Weight array
// is a mask of weights that are applied to surrounding neurons during
// the potential calculation.  The array is of size 2 * LinkingField + 1.
//
//*************************************************************************
  void InitializeWeights()
  {
    linking_weights = new A3<float>(LinkingRadius * 2 + 1,
                                    LinkingRadius * 2 + 1,
                                    LinkingRadius * 2 + 1);
  }

//*************************************************************************
//
// InitializeResults instanitates a new instance of A3.  The results
// array will contain the timestep at which a particular neuron fires.
//
//*************************************************************************
  void InitializeResults(int rows, int cols, int depth, float value)
  {
    results = new A3<float>(rows, cols, depth);
```

```
    results->SetData(value);
  }

//*************************************************************************
//
// InitializeStack instantiates a new stack to operate on neurons and
// sets the maximum number of items that the stack can hold.
//
//*************************************************************************
  void InitializeStack()
  {
    PCNNStack = new Stack<neuron>;
  }

//*************************************************************************
//
// InitializePCNN calls the methods to allocate space for the various
// classes used by the pcnn as well as setting their initial values.
// The input is also normalized at this time.  Therefore, the input must
// be set prior to execution of the pcnn.  This method should be private.
//
//*************************************************************************
  void InitializePCNN()
  {
    int row, col, dep;

    row = input->GetActualRows();
    col = input->GetActualCols();
    dep = input->GetActualDepth();

    InitializeResults(row, col, dep, 0.0);
    InitializeLinkingField(row, col, dep, 0.0);
    InitializeWeights();
    LoadWeights();
    InitializeStack();
    NormalizeInput(input);
  //  InitializeTheta(row, col, dep, GetThetaV());
    Theta1 = 1.0;

  } // end InitializePCNN

//*************************************************************************
//
// SetLinkingRadius assigns the passed value to LinkingRadius.
//
//*************************************************************************
  void SetLinkingRadius(int value)
  {
    LinkingRadius = value;
  }

//*************************************************************************
//
// GetLinkingRadius returns the current value of LinkingRadius
//
//*************************************************************************
```

```
int GetLinkingRadius()
{
  return LinkingRadius;
}

//*************************************************************************
//
// ShowLinkingRadius outputs the current LinkingRadius to Standard Out
//
//*************************************************************************
void ShowLinkingRadius()
{
  cout << "LinkingRadius: " << LinkingRadius << endl;
}

//*************************************************************************
//
// SetLinkingTimeConstant assigns the passed value to LinkingTimeConstant
//
//*************************************************************************
void SetLinkingTimeConstant(float value)
{
  LinkingTimeConstant = value;
}

//*************************************************************************
//
// GetLinkingTimeConstant returns the current value of LinkingTimeConstant
//
//*************************************************************************
float GetLinkingTimeConstant()
{
  return LinkingTimeConstant;
}

//*************************************************************************
//
// ShowLinkingTimeConstant outputs the current LinkingTimeConstant to
// Standard Out.
//
//*************************************************************************
void ShowLinkingTimeConstant()
{
  cout << "LinkingTimeConstant: " << LinkingTimeConstant << endl;
}

//*************************************************************************
//
// SetThetaTimeConstant assigns the passed value to ThetaTimeConstant
//
//*************************************************************************
void SetThetaTimeConstant(float value)
{
  ThetaTimeConstant = value;
}
```

```
//*************************************************************************
//
// GetThetaTimeConstant returns the current value of ThetaTimeConstant
//
//*************************************************************************
  float GetThetaTimeConstant()
  {
    return ThetaTimeConstant;
  }

//*************************************************************************
//
// ShowThetaTimeConstant outputs the current ThetaTimeConstant to
// Standard Out
//
//*************************************************************************
  void ShowThetaTimeConstant()
  {
    cout << "ThetaTimeConstant: " << ThetaTimeConstant << endl;
  }

//*************************************************************************
//
// SetThetaV sets ThetaV to the passed value
//
//*************************************************************************
  void SetThetaV(float value)
  {
    ThetaV = value;
  }

//*************************************************************************
//
// GetThetaV returns the current ThetaV to the calling method
//
//*************************************************************************
  float GetThetaV()
  {
    return ThetaV;
  }

//*************************************************************************
//
// ShowThetaV outputs the current ThetaV to Standard Out
//
//*************************************************************************
  void ShowThetaV()
  {
    cout << "ThetaV: " << ThetaV << endl;
  }

//*************************************************************************
//
// SetTheta0 assigns the passed value to Theta0
//
//*************************************************************************
```

```
  void SetTheta0(float value)
  {
    Theta0 = value;
  }

//*************************************************************************
//
// GetTheta0 returns the current Theta0 to the calling method
//
//*************************************************************************
  float GetTheta0()
  {
    return Theta0;
  }

//*************************************************************************
//
// ShowTheta0 displays the current Theta0 to Standard Out
//
//*************************************************************************
  void ShowTheta0()
  {
    cout << "Theta0: " << Theta0 << endl;
  }

//*************************************************************************
//
// SetLinkingV assigns the passed value to LinkingV
//
//*************************************************************************
  void SetLinkingV(float value)
  {
    LinkingV = value;
  }

//*************************************************************************
//
// GetLinkingV returns the current LinkingV to the calling method
//
//*************************************************************************
  float GetLinkingV()
  {
    return LinkingV;
  }

//*************************************************************************
//
// ShowLinkingV displays the current LinkingV to Standard Out
//
//*************************************************************************
  void ShowLinkingV()
  {
    cout << "LinkingV: " << LinkingV << endl;
  }

//*************************************************************************
```

```
//
// SetBeta assigns the passed value to Beta
//
//*************************************************************************
  void SetBeta(float value)
  {
    Beta = value;
  }

//*************************************************************************
//
// GetBeta returns the current Beta to the calling method
//
//*************************************************************************
  float GetBeta()
  {
    return Beta;
  }

//*************************************************************************
//
// ShowBeta displays the current Beta to Standard Out
//
//*************************************************************************
  void ShowBeta()
  {
    cout << "Beta: " << Beta << endl;
  }

//*************************************************************************
//
// SetPulseHeight assigns the passed value to PulseHeight
//
//*************************************************************************
  void SetPulseHeight(float value)
  {
    PulseHeight = value;
  }

//*************************************************************************
//
// GetPulseHeight returns the current PulseHeight to the calling method
//
//*************************************************************************
  float GetPulseHeight()
  {
    return PulseHeight;
  }

//*************************************************************************
//
// ShowPulseHeight displays the current PulseHeight to Standard Out
//
//*************************************************************************
  void ShowPulseHeight()
  {
```

```
    cout << "PulseHeight: " << PulseHeight << endl;
  }

//*************************************************************************
//
// SetRedraw assigns the passed value to Redraw indicating whether
// neurons fired during the current timestep.
//
//*************************************************************************
  void SetRedraw(int value)
  {
    Redraw = value;
  }

//*************************************************************************
//
// GetRedraw returns the current value of Redraw to the calling method
// indicating whether neurons fired during the current timestep
//
//*************************************************************************
  int GetRedraw()
  {
    return Redraw;
  }

//*************************************************************************
//
// SetTimeSteps assigns the passed value to TimeSteps, indicating the
// number of timesteps to run the pcnn
//
//*************************************************************************
  void SetTimeSteps(int value)
  {
    TimeSteps = value;
  }

//*************************************************************************
//
// GetTimeSteps returns the number of timesteps the pcnn has been set to
// run.
//
//*************************************************************************
  int GetTimeSteps()
  {
    return TimeSteps;
  }

//*************************************************************************
//
// ShowTimeSteps displays the number of timesteps to Standard Out that
// the pcnn is set to run.
//*************************************************************************
  void ShowTimeSteps()
  {
    cout << "TimeSteps: " << TimeSteps << endl;
  }
```

```
//*************************************************************************
//
// SetTimer assigns the passed value to Timer.  Timer holds the current
// timestep of the pcnn
//
//*************************************************************************
  void SetTimer(int value)
  {
    Timer = value;
  }

//*************************************************************************
//
// GetTimer returns the current value of the Timer, which is the current
// timestep of the pcnn, to the calling method
//
//*************************************************************************
  int GetTimer()
  {
    return Timer;
  }

//*************************************************************************
//
// ShowTimer displays to Standard Out the current value of the Timer.
//
//*************************************************************************
  void ShowTimer()
  {
    cout << "Timer: " << Timer << endl;
  }

//*************************************************************************
//
// SetNumClasses assigns the passed value to NumClasses which tracks how
// many of the timesteps produced a pulsing neuron.
//
//*************************************************************************
  void SetNumClasses(int value)
  {
    NumClasses = value;
  }

//*************************************************************************
//
// GetNumClasses returns the number of timesteps that have produced a
// pulsing neuron.
//
//*************************************************************************
  int GetNumClasses()
  {
    return NumClasses;
  }

//*************************************************************************
```

```
//
// ShowNumClasses displays the current number of timesteps that have
// produced a pulsing neuron.  Display is to Standard Out.
//
//*************************************************************************
  void ShowNumClasses()
  {
    cout << "Number of Classes: " << NumClasses << endl;
  }

//*************************************************************************
//
// ShowInput displays to Standard Out the input assigned to the pcnn.
//
//*************************************************************************
  void ShowInput()
  {
    input->ShowA3();
  }

//*************************************************************************
//
// ShowWeights displays to Standar Out the weights used by the pcnn.
//
//*************************************************************************
  void ShowWeights()
  {
    linking_weights->ShowA3();
  }

//*************************************************************************
//
// ShowResults displays to Standard Out the results of the pcnn.  Results
// contains the timestep that a neuron pulsed during execution of the
// pcnn.
//
//*************************************************************************
  void ShowResults()
  {
    results->ShowA3();
  }

//*************************************************************************
//
// ShowTheta displays to Standard Out the values assigned to each of the
// thetas.
//
//*************************************************************************
  void ShowTheta()
  {
    Theta->ShowA3();
  }

//*************************************************************************
//
// GetMaxInput returns the maximum value assigned to the input
```

```
//
//*************************************************************************
 float GetMaxInput()
 {
   return input->GetMax();
 }

//*************************************************************************
//
// ShiftInput shifts the input data by the amount passed
//
//*************************************************************************
 ShiftInput(float amount)
 {
   int dep, row, col;

   for (dep = 0; dep < input->GetActualDepth(); dep++)
     for (row = 0; row < input->GetActualRows(); row++)
       for (col = 0; col < input->GetActualCols(); col++)
         input->Set(dep, row, col, (input->Get(dep, row, col) + amount) );
 }

//*************************************************************************
//
// NormalizeInput changes the input to a range of 0-1.
//
//*************************************************************************
 void NormalizeInput(A3<float> *temp)
 {
   int dep, row, col;
   float factor = 1 / temp->FindMax();
   for (dep = 0; dep < temp->GetActualDepth(); dep++)
     for (row = 0; row < temp->GetActualRows(); row++)
       for (col = 0; col < temp->GetActualCols(); col++)
         temp->Set(dep, row, col, (temp->Get(dep, row, col) * factor) );
 }

//*************************************************************************
//
// LoadWeights initializes the Weight array with values that correspond
// to distances from the center of the weight mask.
//
//*************************************************************************
 void LoadWeights()
 {
   float DistanceSquared;
   int j, k, l;

   for(j = -LinkingRadius; j <= LinkingRadius; j++)
     for(k =- LinkingRadius; k <= LinkingRadius; k++)
       for(l =- LinkingRadius; l <= LinkingRadius; l++)
       {
         DistanceSquared = (float)(j * j + k * k + l * l);
         if(DistanceSquared > 0)
           linking_weights->Set(j + LinkingRadius, k + LinkingRadius,
                                l + LinkingRadius, 1);
```

```
        else
          linking_weights->Set(j + LinkingRadius, k + LinkingRadius,
                               l + LinkingRadius, 0.0);
      } // end for
  } // end LoadWeights

//***************************************************************************
//
// ApplyDecay exponentially decays the linking and theta fields
//
//***************************************************************************
  void ApplyDecay()
  {
    float LinkingDecayRate;
    float ThetaDecayRate;
    int dep, row, col;
    static int ActualRows  = LinkingField->GetActualRows();
    static int ActualCols  = LinkingField->GetActualCols();
    static int ActualDepth = LinkingField->GetActualDepth();

    if (LinkingTimeConstant == 0.0)
      LinkingDecayRate = 0.0;
    else
      LinkingDecayRate = exp(-1 / LinkingTimeConstant);
    if (ThetaTimeConstant == 0.0)
      ThetaDecayRate = 0.0;
    else
      ThetaDecayRate = exp(-1 / ThetaTimeConstant);
    Theta1 = Theta1 * ThetaDecayRate;
    for (dep = 0; dep < ActualDepth; dep++)
      for (row = 0; row < ActualRows; row++)
        for (col = 0; col < ActualCols; col++)
        {
          if (LinkingRadius > 0)
            LinkingField->Set(dep, row, col, LinkingField->Get(dep, row, col)
                                             * LinkingDecayRate);
    //      if (Theta->Get(dep, row, col) < 100)
    //        Theta->Set(dep, row, col, Theta->Get(dep, row, col)
    //                                  * ThetaDecayRate);
        } // end for
  } // end ApplyDecay

//***************************************************************************
//
// UpdateLinkingAndThetaInternalState updates the theta field of the
// firing neuron and the linking field of the surrounding neurons.
//
//***************************************************************************
  void UpdateLinkingAndThetaInternalState(int dep, int row, int col)
  {
    int surdep, surrow, surcol;
    int wdep, wrow, wcol;
    float temp_fired = LinkingField->Get(dep, row, col);

    if (LinkingRadius > 0)
      for (surdep = dep - LinkingRadius, wdep = LinkingRadius * 2;
```

```
        wdep >= 0; wdep--, surdep++)
      for (surrow = row - LinkingRadius, wrow = LinkingRadius * 2;
           wrow >= 0; wrow--, surrow++)
        for (surcol = col - LinkingRadius, wcol = LinkingRadius * 2;
             wcol >= 0; wcol--, surcol++)
          if (InBounds(surdep, surrow, surcol))
          {
            LinkingField->Set(surdep, surrow, surcol,
                              LinkingField->Get(surdep, surrow, surcol) +
                              LinkingV * linking_weights->Get(wdep, wrow, wcol));
          }
//    Theta->Set(dep, row, col, Theta->Get(dep, row, col) + ThetaV *
//               PulseHeight);
    LinkingField->Set(dep, row, col, temp_fired);
  } // end UpdateLinkingAndThetaInternalState

//*************************************************************************
//
// CanNeuronFire calculates the Uk of the given neuron and determines
// whether the neuron exceeds the threshold.  If the neuron can fire,
// the neuron is pushed on the stack, results for that neuron are set.
//
//*************************************************************************
  void CanNeuronFire(int dep, int row, int col)
  {
    float  Lk     = 0.0;
    float  Uk     = 0.0;
    float  Thresh = 0.0;
    neuron CurrentNeuron = {dep, row, col};


    // if (Uk > Theta)
    Uk = input->Get(dep, row, col) * (1 + Beta
                    LinkingField->Get(dep,row,col));
    //  Thresh = Theta0 + Theta->Get(dep, row, col);
    Thresh = Theta0 + Theta1;
    if (Uk  > Thresh)
    {
      SetRedraw(1);
      PCNNStack->push(CurrentNeuron);
      results->Set(dep, row, col, (float) GetTimer());
      UpdateLinkingAndThetaInternalState(dep, row, col);
      CheckNeighbors();
    } // end if (Uk > Theta)
  } // end CanNeuronFire

//*************************************************************************
//
// CheckNeighbors checks the neighbors surrounding the given neuron to
// see if they can fire.
//
//*************************************************************************
  void CheckNeighbors()
  {
    neuron CurrentNeuron;
    int surdep, surrow, surcol;
```

```
    float Uk, Thresh, Lk = 0.0;

    while (!PCNNStack->IsEmpty())          // check surrounding neurons
    {
      CurrentNeuron = PCNNStack->pop();
      for (surdep = CurrentNeuron.dep - LinkingRadius;
           surdep <= CurrentNeuron.dep + LinkingRadius; surdep++)
        for (surrow = CurrentNeuron.row - LinkingRadius;
             surrow <= CurrentNeuron.row + LinkingRadius; surrow++)
          for (surcol = CurrentNeuron.col - LinkingRadius;
               surcol <= CurrentNeuron.col + LinkingRadius; surcol++)
            if (results->Get(surdep, surrow, surcol) < 1.0 &&
                InBounds(surdep, surrow, surcol) )
            {
              Lk = LinkingField->Get(surdep, surrow, surcol);
              // if (Uk > Theta)
              Uk = input->Get(surdep, surrow, surcol) * (1 + Beta * Lk);
              //  Thresh = Theta0 + Theta->Get(surdep, surrow, surcol);
              Thresh = Theta0 + Theta1;
              if ( Uk > Thresh)
              {
                neuron PushNeuron = {surdep, surrow, surcol};
                PCNNStack->push(PushNeuron);
                results->Set(surdep, surrow, surcol, (float) GetTimer());
                UpdateLinkingAndThetaInternalState(surdep, surrow, surcol);
              } // end if (Uk > Theta)
            } // end if
    } // end while
  } // end CheckNeighbors

//************************************************************************
//
// InBounds determines whether the given neuron is within the range of
// the image/volume
//
//************************************************************************
  inline int InBounds(int dep, int row, int col)
  {
    static int ActualDepth = input->GetActualDepth();
    static int ActualRows  = input->GetActualRows();
    static int ActualCols  = input->GetActualCols();

    if ( (dep < 0) || (row < 0) || (col < 0) || (dep >= ActualDepth) ||
         (row >= ActualRows) || (col >= ActualCols) )
      return 0;
    else
      return 1;
  } // end InBounds

//************************************************************************
//
// CalculateLinkingAndOutput calls CanNeuronFire for every neuron that
// has not fired yet.
//
//************************************************************************
  void CalculateLinkingAndOutput()
```

```
  {
    int dep, row, col;

    for (dep = 0; dep < input->GetActualDepth(); dep++)
      for (row = 0; row < input->GetActualRows(); row++)
        for (col = 0; col < input->GetActualCols(); col++)
          if (results->Get(dep, row, col) < 1.0)
            CanNeuronFire(dep, row, col);
  } // end CalculateLinkingAndOutput

//*************************************************************************
//
// Exponentialize returns the normalized values to the range of the
// original input.
//
//*************************************************************************
  void Exponentialize()
  {
    int dep, row, col;
    float max = input->GetMax();
     for (dep = 0; dep < results->GetActualDepth(); dep++)
       for (row = 0; row < results->GetActualRows(); row++)
         for (col = 0; col < results->GetActualCols(); col++)
           if (results->Get(dep, row, col) >= 1.0)
             results->Set(dep, row, col, (float) (max *
               exp(-(results->Get(dep, row, col)-1) / ThetaTimeConstant)));
    results->SetMax(results->FindMax());
  }

//*************************************************************************
//
// ZeroOneMap converts the timestep results to a zero one map
//
//*************************************************************************
  void ZeroOneMap()
  {
    int dep, row, col;
    for (dep = 0; dep < results->GetActualDepth(); dep++)
      for (row = 0; row < results->GetActualRows(); row++)
        for (col = 0; col < results->GetActualCols(); col++)
          if (results->Get(dep, row, col) > 0.0)
            results->Set(dep, row, col, 1.0);
  }

//*************************************************************************
//
// OneEpoch performs one run of the pcnn with the number of timesteps set.
//
//*************************************************************************
  void OneEpoch()
  {
    int TimeCounter;

    SetNumClasses(0);
    cout << "TimeStep: " << flush;
    for (TimeCounter = 1; TimeCounter <= TimeSteps; TimeCounter++)
```

```
    {
      SetTimer(TimeCounter);
      SetRedraw(0);
      ApplyDecay();
      cout << TimeCounter << " " << flush;
      CalculateLinkingAndOutput();
      if (GetRedraw()) // Something Pulsed this time step
      {
        cout << ". ";
        SetNumClasses(GetNumClasses() + 1);
      }
    } // end for TimeCounter
    cout << endl;
  } // end OneEpoch

//************************************************************************
//
// Destructor calls destructors of the classes used by the pcnn.
//
//************************************************************************
  ~PCNN()
  {
    delete results;
    delete Theta;
    delete linking_weights;
  } // end Destructor

};
#endif
```

D.2 PCNN Segmenter Code

```
/*************************************************************
*
* Class PCNN_Segmenter contains methods specific to a PCNN
* used to perform segmentation.
*
* The variables required by the PCNN base class MUST be set
* prior to use.s
*
/*************************************************************

#include "PCNN4.cc"

class PCNN_Segmenter : public PCNN{

public:

//************************************************************************
//
// Constructor initializes all variables of the pcnn segmenter to default
// values
//
//************************************************************************
```

```
  PCNN_Segmenter()
    : PCNN()
  {}

//************************************************************************
//
// ExecuteSegmenter executes the pcnn (one epoch) to perform segmentation.
//
//************************************************************************
  A3<float>* ExecuteSegmenter()
  {
    time_t t = time(NULL);
    OneEpoch();
    if (output == Intensity)
      Exponentialize();
    cout << "Program took: " << time(NULL) - t << " seconds" << endl;
    ShowNumClasses();
    return results;
  } // end ExecuteSegmenter

//************************************************************************
//
// GetOutput returns the result of the pcnn to the calling method.  If
// the pcnn has not been executed, it is executed, otherwise, results is
// returned.
//
//************************************************************************
  A3<float>* GetOutput()
  {
    if (results == NULL)
    {
      InitializePCNN();
      return ExecuteSegmenter();
    }
    return results;
  }

};
```

D.3 PCNN Filter Code

```
/*************************************************************
*
* Class PCNN_Filter is a subclass of PCNN and contains methods
* and variables specific to performing PCNN filtering.
*
* The following MUST be set prior to use:
*
*   Those variables required by the PCNN
*   FilterThreshPercent
*
/*************************************************************

#include "PCNN4.cc"
```

```
class PCNN_Filter : public PCNN{

protected:

  float          FilterThreshPercent;

public:

//************************************************************************
//
// Constructor initializes all variables of the pcnn filter to default
// values
//
//************************************************************************
  PCNN_Filter()
    : PCNN(),
      FilterThreshPercent(0)
  {}


//************************************************************************
//
// ExecuteFilter executes a pcnn filter.  The pcnn is executed until the
// desired threshold is reached.
//
//************************************************************************
  A3<float>* ExecuteFilter()
  {
    int num_adjusts = results->GetActualCols() * results->GetActualRows() *
                      results->GetActualDepth();
    int num_epochs = 0;

    time_t t = time(NULL);

    while ( results->GetActualCols() * results->GetActualRows() *
          results->GetActualDepth() * GetFilterThreshPercent()  <= num_adjusts)
    {
      results->SetData(0.0);
      //  Theta->SetData(GetThetaV());
      Theta1 = 1.0;
      LinkingField->SetData(0.0);
      OneEpoch();
      num_epochs = num_epochs + 1;
      num_adjusts = AdjustValues();
      cout << "Epoch: " << num_epochs << endl;
      cout << "Number of Adjustments: " << num_adjusts << endl;

    }
    Exponentialize();
    cout << "Program took: " << time(NULL) - t << " seconds" << endl;
    cout << "Number of Classes: " << GetNumClasses() << endl;
    return results;
  }

//************************************************************************
```

```
//
// Check Majority
//
//*************************************************************************
  CheckMajority(int dep, int row, int col)
  {
    int   surdep, surrow, surcol, before = 0, after = 0;
    float target = results->Get(dep, row, col);
    int   neighbors = (LinkingRadius * 2 + 1) * (LinkingRadius * 2 + 1) - 1;
    int   half;

    if (results->GetActualDepth() > 1)
      neighbors = (int) pow((LinkingRadius * 2 + 1), 3) - 1;

    half = neighbors / 2;

    for (surdep = dep - LinkingRadius; surdep <= dep + LinkingRadius; surdep++)
      for (surrow = row - LinkingRadius; surrow <= row + LinkingRadius;
           surrow++)
        for (surcol = col - LinkingRadius; surcol <= col + LinkingRadius;
             surcol++)
          if (InBounds(surdep, surrow, surcol))
          {
            if (results->Get(surdep, surrow, surcol) > target)
              after =  after + 1;
            else if (results->Get(surdep, surrow, surcol) < target)
              before = before + 1;
          }
    if (before > half)
      return (1);
    else if (after > half)
      return (-1);
    else
      return (0);

  }

//*************************************************************************
//
// AdjustValues is called by the filter and adjusts the intensities of
// pixels based on the firing order of the neurons around it.  If a
// majority of neurons fire before, then the intensity is adjusted
// upward.  If a majority fire after, the intensity is adjusted
// downward.  If a majority fire at the same time, no action is taken.
//
//*************************************************************************
  int AdjustValues()
  {
    int   dep, row, col, count = 0;
    static float factor = 1 / input->GetMax();
    int   direction;

    for (dep = 0; dep < results->GetActualDepth(); dep++)
      for (row = 0; row < results->GetActualRows(); row++)
        for (col = 0; col < results->GetActualCols(); col++)
          if (results->Get(dep, row, col) > 0.0)
```

```
        {
          direction = CheckMajority(dep, row, col);
          if (direction != 0)
          {
            input->Set(dep, row, col, input->Get(dep, row, col) +
                       factor * direction);
            count++;
          }
        }
    return count;
  }

//************************************************************************
//
// SetFilterThreshPercent assigns the passed value to FilterThreshPercent
//
//************************************************************************
  void SetFilterThreshPercent(float value)
  {
    FilterThreshPercent = value;
  }

//************************************************************************
//
// GetFilterThreshPercent returns the current FilterThreshPercent to the
// calling method.
//
//************************************************************************
  float GetFilterThreshPercent()
  {
    return FilterThreshPercent;
  }

//************************************************************************
//
// GetOutput returns the result of the pcnn to the calling method.  If
// the pcnn has not been executed, it is executed, otherwise, results is
// returned.
//
//************************************************************************
  A3<float>* GetOutput()
  {
    if (results == NULL)
    {
      InitializePCNN();
      return ExecuteFilter();
    }
    return results;
  }

};
```

```
/*************************************************************
*
* Class PCNN_Pulser contains all the methods specific to
* a PCNN that allows neurons to pulse multiple times.
*
* The variables required by the PCNN base class MUST be set
* prior to use.
*
/*************************************************************

#include "PCNN4.cc"

class PCNN_Pulser : PCNN{

public:

//************************************************************************
//
// Constructor initializes all variables of the pcnn pulser to default
// values
//
//************************************************************************
  PCNN_Pulser()
    : PCNN()
  {}

//************************************************************************
//
// ExecutePulser executes one timestep each time it is called and returns
// a zero/one map of the neurons that pulsed at that timestep.
//
//************************************************************************
  A3<float>* ExecutePulser()
  {
    results->SetData(0.0);
    SetPulseHeight(1.0);
    SetTimer(GetTimer() + 1);
    SetRedraw(0);
    ApplyDecay();
    CalculateLinkingAndOutput();
    if (GetRedraw()) // Something Pulsed this time step
    {
      SetNumClasses(GetNumClasses() + 1);
      ZeroOneMap();
    }
    return results;
  }

//************************************************************************
//
// GetOutput returns the result of the pcnn to the calling method.  If
// the pcnn has not been executed, it is executed, otherwise, results is
// returned.
//
```

```
//*************************************************************************
  A3<float>* GetOutput()
  {
    if (results == NULL)
    {
      InitializePCNN();
      return ExecutePulser();
    }
    return ExecutePulser;
  }

};
```

D.5 PCNN Reader Code

```
/*************************************************************
*
* Class Reader performs a file read.  It requires that the
* file be represented as 16 bit binary values.
*
* The following variables MUST be set prior to use:
*   ActualRows
*   ActualCols
*   ActualDepth
*   ImageLow
*   ImageHigh
*   FilePrefix
*
/*************************************************************

#include "A3.cc"

class Reader{

protected:

  int ActualRows;
  int ActualCols;
  int ActualDepth;
  int ImageLow;
  int ImageHigh;
  char *FilePrefix;
  A3<float> *Volume;

public:

  // Constructor
  Reader()
    : Volume(NULL),
      ActualRows(0),
      ActualCols(0),
      ActualDepth(0),
      ImageLow(0),
      ImageHigh(0),
```

```
    FilePrefix(NULL)
  {}

  // Destructor
  ~Reader()
  {
    delete Volume;
  }

  void SetDataDimensions(int rows, int cols, int depth)
  {
    ActualRows  = rows;
    ActualCols  = cols;
    ActualDepth = depth;
  }

  void SetFilePrefix(char *InFilename)
  {
    FilePrefix = InFilename;
  }

  void SetImageRange(int low, int high)
  {
    ImageLow  = low;
    ImageHigh = high;
  }

//*************************************************************************
//
// Execute reads in a series of images and fills the 3D input array.
// The method also works with 1 and 2 dimensional data as well.
//
//*************************************************************************
  void Execute()
  {
    short int *hold;
    int       dep, row, col;
    FILE      *FID;
    char      new_filename[100];
    size_t    result;
    //fill a 3D matrix

    if ( (!ActualRows) || (!ActualCols) || (!ActualDepth) )
    {
      cout << "Data Dimensions not set properly...exiting!" << endl;
      exit(-1);
    }

    Volume = new A3<float> (ActualRows, ActualCols, ActualDepth);

    for(dep = 0; dep < ActualDepth; dep++)
    {
      sprintf(new_filename, "%s.%d", FilePrefix, dep + ImageLow);
      printf("Loading binary R.F. file %s: \n", new_filename);

      FID = fopen(new_filename, "rb");
```

```
      if (FID == NULL)
      {
        printf("\nCould not open input file %s ...exiting\n", new_filename);
        exit(-1);
      } // end if FID

      hold = new short int[ActualCols];

      for(row = 0; row < ActualRows; row++)
      {
        result = fread(hold, 2, ActualCols, FID);
        if(ActualCols != result)
        {
          printf("\nError while reading binary R.F. input file %s...exiting\n",
                 new_filename);
          exit(-1);
        } // end if != result
        for(col = 0; col < ActualCols; col++)
          Volume->Set(dep, row, col, (float)hold[col]);
      } // end for row
      fclose(FID);
    } // end for dep
    delete[] hold;
  } // Execute

  A3<float>* GetOutput()
  {
    if (Volume == NULL)
    {
      Execute();
      Volume->SetMax(Volume->FindMax());
    }
    return Volume;
  }
};
```

D.6 PCNN Writer Code

```
/*****************************************************************
*
* Class Writer performs a file write.  Output is 16 binary
* values.
*
* The following variables MUST be set prior to use:
*   FilePrefix
*   Input
*
/*****************************************************************

#include "A3.cc"

class Writer{

protected:
```

```
  A3<float>     *Input;
  char          *FilePrefix;

public:

  // Constructor
  Writer()
    : FilePrefix(NULL),
      Input(NULL)
  {}

  // Destructor
  ~Writer()
  {}

  void SetInput(A3<float>* in_volume)
  {
    Input = in_volume;
  }

  void SetFilePrefix(char *OutFilename)
  {
    FilePrefix = OutFilename;
  }
//************************************************************************
//
// Execute writes the data to file.  Works with 1 and 2 dimensional data
// as well.
//
//************************************************************************
  void Execute()
  {
    FILE       *fid;
    int        row, dep, col;
    char       new_filename[100];
    size_t     result;
    int        ActualRows = Input->GetActualRows();
    int        ActualCols = Input->GetActualCols();
    int        ActualDepth = Input->GetActualDepth();
    short int  temp;
    static int file_extension = 0;

    for (dep = 0; dep < ActualDepth; dep++)
    {
      sprintf(new_filename, "%s.%d", FilePrefix, file_extension);
      cout << "Writing binary file: " << new_filename << endl;

      fid = fopen(new_filename, "w");
      if (fid == NULL)
      {
        printf("\nCould not open input file %s ...exiting\n", new_filename);
        exit(-1);
      } // end if FID

      for (row = 0; row < ActualRows; row++)
```

```
        for (col = 0; col < ActualCols; col++)
        {
          temp = (short int) Input->Get(dep, row, col);
          fwrite(&temp, sizeof(short int), 1, fid);
        }

      fclose(fid);
      file_extension++;
    }
  }
};
```

D.7 PCNN Stack Code

```
/*************************************************************
*
* Class NewStack implements a stack using a template.  The
* stack is represented by a linked list.
*
/*************************************************************

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <memory.h>
#include <ctype.h>
#include <time.h>
#include <values.h>
#include <iostream.h>

//************************************************************
//
//Class Stack is a generic stack that accepts the data type
//which the stack was instantiated with.  Operations are push
//and pop.
//
//************************************************************

template <class STACKTYPE>
class Stack{

protected:
    typedef struct stack_rec{
      STACKTYPE  item;
      stack_rec* next;
    };
    stack_rec* current;

public:

  //constructor
  Stack()
```

```
  {
    current = NULL;
  }

  //destructor
  ~Stack()
  {
  }

  inline void push(STACKTYPE item)
  {
    stack_rec* temp;
    temp = new stack_rec;
    temp->item = item;
    temp->next = current;
    current = temp;
  }

  inline STACKTYPE pop()
  {
    STACKTYPE temp;
    stack_rec* old;
    if (current != NULL)
    {
       old = current;
       temp = current->item;
       current = current->next;
       delete old;
       return(temp);
    }
    else
    {
      cerr << "Stack is empty" << endl;
      exit(-1);
    }
  }

  inline int IsEmpty()
  {
    if (current == NULL)
      return 1;
    else
      return 0;
  }
};
```

D.8 PCNN A3 Code

```
/*****************************************************************
*
* Class A3 is a container for a three dimensional array.  It is
* implemented as a template.  By setting depth to 1, a two
* dimensional array is created.  By setting depth and columns
* to 1, a one dimensional array is created.
```

```
*
* The following parameters MUST be set prior to use:
*   ActualRows
*   ActualCols
*   ActualDepth
*
* They are set when the class constructor is called
*
/*************************************************************

#ifndef __A3__
#define __A3__

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <memory.h>
#include <ctype.h>
#include <time.h>
#include <values.h>
#include <iostream.h>

/*************************************************************
*  Matrix_3_D creates a three dimensional matrix for storing
*  a volume constructed from a series of images.
*  ActualRows and ActualCols are the size of an image
*  ActualDepth is the number of images
*
*************************************************************/
template <class TYPE>
class A3{

protected:
  TYPE*** matrix;
  int ActualRows;
  int ActualCols;
  int ActualDepth;
  TYPE max;
  TYPE min;

public:

  A3(int p_ActualRows, int p_ActualCols, int p_ActualDepth)
    : matrix(NULL),
      ActualRows(p_ActualRows),
      ActualCols(p_ActualCols),
      ActualDepth(p_ActualDepth)
  {
    int dep, row, col;

    matrix = (TYPE ***) calloc(ActualDepth,sizeof(TYPE **));
    if (matrix == NULL)
      {
        printf("Not enough memory to hold 3D matrix. . .exiting\n");
```

```
      exit(-1);
    }

  for(dep = 0; dep < ActualDepth; dep++)
    {
      matrix[dep] = (TYPE **) calloc(ActualRows,sizeof(TYPE*));//Cols
      if (matrix[dep] == NULL)
        {
          printf("Not enough memory to hold 3D matrix. . .exiting\n");
          exit(-1);
        }
      for (row = 0; row < ActualRows; row++) //col
        {
          matrix[dep][row] = (TYPE*) calloc(ActualCols, sizeof(TYPE));//col
          if (matrix[dep][row] == NULL)//col
            {
              printf("Not enough memory to hold 3D matrix. . .exiting\n");
              exit(-1);
            }
        }
    }
}

~A3()
{
  int dep, row; //col

  if (matrix != NULL)
    {
      for(dep = 0; dep < ActualDepth; dep++)
        for(row = 0; row < ActualRows; row++)//Cols
          if(matrix[dep][row] != NULL)//col
            free(matrix[dep][row]);//col
      if (matrix != NULL)
        free(matrix);
    }
}

inline TYPE*** Matrix()
  {
    return matrix;
  }

inline TYPE Get(int i, int j, int k)
  {
    if ( (i < 0) || (j < 0) || (k < 0) || (i >= ActualDepth) ||
         (j >= ActualRows) || (k >= ActualCols) )
      return (TYPE) 0;
    else
      return matrix[i][j][k];
  }


inline void Set(int i, int j, int k, const TYPE& p_in)
  {
```

```
    matrix[i][j][k] = p_in;
  }

void SetData(const TYPE& p_in)
  {
    int dep, row, col;
    for(dep = 0; dep < ActualDepth; dep++)
      for(row = 0; row < ActualRows; row++)
        for(col = 0; col < ActualCols; col++)
          matrix[dep][row][col] = p_in;
  }


void ShowA3()
  {
    int dep, row, col;
    cout << "Depth: " << ActualDepth << " Rows: " << ActualRows <<
      " Cols: " << ActualCols << endl;
    for(dep = 0; dep < ActualDepth; dep++)
      for(row = 0; row < ActualRows; row++)
        for(col = 0; col < ActualCols; col++)
          cout << "dep: " << dep << " row: " << row
               << " col: " << col << " M: " << matrix[dep][row][col] << endl;
  }

void ShowElement(int dep, int row, int col)
{
  cout << matrix[dep][row][col] << endl;
}


TYPE FindMax()
{
  int dep = 0, row = 0, col = 0;
  TYPE tempmax = matrix[dep][row][col];
  for (dep = 0; dep < ActualDepth; dep++)
    for (row = 0; row < ActualRows; row++)
      for (col = 0; col < ActualCols; col++)
        if (matrix[dep][row][col] > tempmax)
          tempmax = matrix[dep][row][col];
  return tempmax;

}


void FindMin()
{
  int dep, row, col;
  TYPE tempmin = (TYPE) MAXINT;
  for (dep = 0; dep < ActualDepth; dep++)
    for (row = 0; row < ActualRows; row++)
      for (col = 0; col < ActualCols; col++)
        if (matrix[dep][row][col] < tempmin)
          tempmin = matrix[dep][row][col];
  SetMin(tempmin);
}
```

```
  inline void SetMax(const TYPE& p_in)
    {
      max = p_in;
    }

  void SetMin(const TYPE& p_in)
    {
      min = p_in;
    }

  inline TYPE GetMax()
    {
      return max;
    }

  TYPE GetMin()
    {
      return min;
    }

  inline int GetActualRows()
    {
      return ActualRows;
    }

  inline int GetActualCols()
    {
      return ActualCols;
    }

  inline int GetActualDepth()
    {
      return ActualDepth;
    }
};

#endif
```

Bibliography

1. C. Tsai, B. S. Manjunath and R. Jagadeesan. "Automated Segmentation of Brain MR Images," *Pattern Recognition*, *28*(12):1825–1837 (1995).

2. Carlton, R. R. and A. M. Adler. *Principles of Radiographic Imaging*. Delmar, 1996.

3. Clarke, L. P., et al. "MRI Segmentation: Methods and Applications," *Magnetic Resonance Imaging*, *13*(3):343–368 (1995).

4. Ge, Y., et al. "Accurate Localization of Cortical Convolutions in MR Brain Images," *IEEE Transactions on Medical Imaging*, *15*(4):418–428 (1996).

5. Hall, L. O., et al. "A Comparison of Neural Network and Fuzzy Clustering Techniques in Segmenting Magnetic Resonance Images of the Brain," *IEEE Transactions on Neural Networks*, *3*(5):672–682 (1992).

6. Lorensen, W. E. and H. E. Cline. "Marching Cube: A High Resolution 3-D Surface Construction Algorithm," *Computer Graphics*, *21*(3):163–169 (1987).

7. M. Ozkan, B. M. Dawant and R. J. Maciunas. "Neural-Network-Based Segmentation of Multi-Modal Medical Images: A Comparative and Prospective Study," *IEEE Transactions on Medical Imaging*, *12*(3):534–544 (1993).

8. M. Sonka, S. K. Tadikonda and S. M. Collins. "Knowledge-Based Interpretation of MR Brain Images," *IEEE Transactions on Medical Imaging*, *15*(4):443–452 (1996).

9. R. Eckhorn, H. J. Reitboeck, M. Arndt and P. Dicke. "Feature Linking via Synchronization among Distributed Assemblies: Simulations of Results from Cat Visual Cortex," *Neural Computation*, 293–307 (1990).

10. Ranganeth, H. S. and G. Kuntimad. "Pulse Coupled Neural Networks for Image Processing." Obtained from Capt Randy Broussard.

11. Rumbaugh, J., et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

12. S. C. Amartur, D. Piraino and Y. Takefuji. "Optimization Neural Networks for the Segmentation of Magnetic Resonance Images," *IEEE Transactions on Medical Imaging*, *11*(2):215–220 (1992).

13. W. M. Wells, W. E. L. Grimson, R. Kikinis and F. A. Jolesz. "Adaptive Segmentation of MRI Data," *IEEE Transactions on Medical Imaging*, *15*(4):429–442 (1996).

14. W. Schroeder, K. Martin and W. E. Lorensen. *The Visualization Toolkit*. Prentice Hall, 1996.

15. Worth, A. J. and D. N. Kennedy. "Segmentation of Magnetic Resonance Brain Images Using Analogue Constraint Satisfaction Neural Networks," *Image and Vision Computing*, *12*(6):345–354 (1994).

Vita

Lieutenant Shane Abrahamson [redacted]. In May 1993 he graduated from Montana State University with a Bachelor of Science degree in Computer Science and was awarded a commission in the United States Air Force. His first assignment was as a Data Standardization Officer at the Air Force Command Control Communications and Computer Agency (AFC4A) at Scott Air Force Base, Illinois. In May 1995, Lieutenant Abrahamson entered the Computer Systems program at the Air Force Institute of Technology.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1996	Master's Thesis

4. TITLE AND SUBTITLE
PULSE COUPLED NEURAL NETWORKS FOR THE SEGMENTATION OF MAGNETIC RESONANCE BRAIN IMAGES

5. FUNDING NUMBERS

6. AUTHOR(S)
Shane L. Abrahamson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
This research develops an automated method for segmenting Magnetic Resonance (MR) brain images based on Pulse Coupled Neural Networks (PCNN). MR brain image segmentation has proven difficult, primarily due to scanning artifacts such as interscan and intrascan intensity inhomogeneities. The method developed and presented here uses a PCNN to both filter and segment MR brain images. The technique begins by preprocessing images with a PCNN filter to reduce scanning artifacts. Images are then contrast enhanced via histogram equalization. Finally, a PCNN is used to segment the images to arrive at the final result. Modifications to the original PCNN model are made that drastically improve performance while greatly reducing memory requirements. These modifications make it possible to extend the method to filter and segment three dimensionally. Volumes represented as series of images are segmented using this new method. This new three dimensional segmentation technique can be used to obtain a better segmentation of a single image or of an entire volume. Results indicate that the PCNN shows promise as an image analysis tool.

14. SUBJECT TERMS
Neural Nets, Image Processing, Magnetic Resonance Imaging, Three Dimensional, Filters, Segmented, Visual Perception, PCNN(Pulse Coupled Neural Networks)

15. NUMBER OF PAGES
91

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.